



varian data machines / a varian subsidiary

VARIAN 620
FORTRAN IV REFERENCE
MANUAL



VARIAN 620
FORTTRAN IV REFERENCE MANUAL

Specifications Subject to Change Without Notice



varian data machines/a varian subsidiary

© 1971



varian data machines

98 A 9902 035

February 1971

98 A 9902 025



TABLE OF CONTENTS

SECTION 1 INTRODUCTION

1.1	General.....	1-1
1.2	Character Set.....	1-2
1.3	Line Format.....	1-3

SECTION 2 DATA

2.1	General.....	2-1
2.2	Data Types.....	2-1
2.3	Data Names.....	2-1
2.4	Variables.....	2-2
2.5	Constants.....	2-2
2.6	Arrays.....	2-7

SECTION 3 SPECIFICATIONS AND STATEMENTS

3.1	General.....	3-1
3.2	Dimension Statement.....	3-1
3.3	Common Statement.....	3-2
3.4	Equivalence Statement.....	3-5
3.5	Type Statement.....	3-6

SECTION 4 EXPRESSIONS AND ASSIGNMENTS

4.1	Arithmetic Expressions.....	4-1
4.2	Arithmetic Assignment Statement.....	4-4
4.3	Logical Assignment Statement.....	4-6
4.4	Logical Expressions.....	4-7



SECTION 5 CONTROL STATEMENTS

5.1	General.....	5-1
5.2	Go To Statements.....	5-1
5.3	Arithmetic If Statement.....	5-4
5.4	Call Statement.....	5-6
5.5	Return Statement.....	5-7
5.6	Continue Statement.....	5-7
5.7	Pause Statement.....	5-8
5.8	Stop Statement.....	5-8
5.9	Do Statement.....	5-9

SECTION 6 INPUT/OUTPUT STATEMENTS

6.1	General.....	6-1
6.2	Input/Output Lists.....	6-1
6.3	Simple Lists.....	6-1
6.4	Do-Implied Lists.....	6-2
6.5	Read Statements.....	6-3
6.6	Write Statements.....	6-4
6.7	Rewind Statement.....	6-4
6.8	Backspace Statement.....	6-5
6.9	Endfile Statement.....	6-5
6.10	Format Statement.....	6-5
6.11	Field Specifications.....	6-6
6.12	F Conversion.....	6-6
6.13	E Conversion.....	6-8
6.14	D Conversion.....	6-9
6.15	I Conversion.....	6-9
6.16	A Conversion.....	6-10
6.17	H Conversion.....	6-10
6.18	X Specification.....	6-12
6.19	L Format Code.....	6-12
6.20	G Format Code.....	6-13
6.21	Scale Factor P.....	6-15
6.22	/ Specification.....	6-18
6.23	Repeat Specifications.....	6-19
6.24	Format Control and Line Interaction.....	6-21



SECTION 7 PROGRAMS AND SUBPROGRAMS

7.1	General.....	7-1
7.2	Main Programs.....	7-1
7.3	Subprograms.....	7-1
7.4	Statement Functions.....	7-8
7.5	Intrinsic Functions.....	7-9
7.6	Basic External Functions.....	7-9
7.7	Dummy Arguments.....	7-9
7.8	Adjustable Dimensions.....	7-10
7.9	External Statement.....	7-12

SECTION 8 OPERATING INSTRUCTIONS

8.1	General.....	8-1
8.2	Compiler Operating Instructions.....	8-1
8.3	Loader Operating Instructions.....	8-2
8.4	Input/Output Device Specifications.....	8-6
8.5	Compiler Input Records.....	8-7
8.6	Compiler Output Records.....	8-8
8.7	Notification Errors.....	8-8
8.8	Termination Errors.....	8-9
8.9	Optional Listings.....	8-9
8.10	Maps.....	8-10
8.11	I/O Device Selection.....	8-13

APPENDIX A STANDARD BCD CODES



LIST OF ILLUSTRATIONS AND TABLES

Figure 1-1	Sample FORTRAN Coding Forms	1-4
Figure 8-1.	Run-Time Map for Stand-Alone FORTRAN IV	8-11
Table 7-1	Intrinsic Functions	7-14
Table 7-2	Basic External Functions	7-16



SECTION 1 INTRODUCTION

Varian **FORTRAN IV** is a programming system for the 620 family of computers and is comprised of a language, a library of subprograms, a compiler, and a run-time package (program). It is available in both stand-alone and operating system (MOS) configurations.

1.1 GENERAL

The **FORTRAN IV** language is especially useful in writing programs for scientific and engineering applications that involve mathematical computations. In fact, the name of the language—**FORTRAN**—is derived from its primary use: **FORM**ula **TRAN**slating. Source programs written in the **FORTRAN** language consist of a set of statements constructed from the elements described in this publication. The **FORTRAN** compiler analyzes the source program statements and transforms them into an object program that is suitable for execution on the 620. In addition, when the **FORTRAN** compiler detects errors in the source program, appropriate error messages are produced. The Varian **FORTRAN IV** language is compatible with and encompasses the American National Standards Institute (ANSI) **FORTRAN** including its mathematical subroutine provisions. Any valid programs compiled and executed using basic **FORTRAN** subset may also be compiled and executed by the **FORTRAN IV** compiler. Equivalent results are ensured by:

- a. Common data formats.
- b. Common format code routines.
- c. Common mathematical subroutines.
- d. Common libraries.

The following **FORTRAN IV** features facilitate the writing of source programs:

- a. Scale Factor: The scale factor allows modification of data during conversion between internal and external representation.
- b. Variable Attribute Control: The attributes of variables and arrays can be explicitly specified in the source program by statements that:
 - (1). Specify the number of words assigned to an item.
 - (2). Explicitly type a variable as integer, real, double precision, complex, or logical.
 - (3). Specify the dimension of arrays.
 - (4). Specify data initialization values for variables.



SECTION 1 INTRODUCTION

- c. Adjustable Array Dimensions: The dimensions of an array in a subprograms can be specified as variables. When the subprogram is called, the absolute array dimensions are substituted.
- d. Three Dimension Arrays: An array has one, two, or three dimensions.
- e. Six Character Variable Names: The name of a variable contains up to six characters.
- f. Function subprograms return results via the argument list.

1.2 CHARACTER SET

A **FORTRAN** program unit is written using the following letters, digits, and special characters:

Letters: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z \$

Digits: 0 1 2 3 4 5 6 7 8 9

Special Characters:

	blank or space
=	equals
+	plus
-	minus
*	asterisk
/	slash
(	left parenthesis
)	right parenthesis
,	comma
.	decimal point

With the exception of the specific uses indicated in the following sections of this manual, a blank character has no meaning, and can be used freely by the programmer to improve the readability of the **FORTRAN** program.

The following special characters are classified as arithmetic operators and are significant in the unambiguous statement of arithmetic expressions:



SECTION 1 INTRODUCTION

+	addition or positive value
-	subtraction or negative value
*	multiplication
/	division
**	exponentiation

The special characters equals (=), open parenthesis ((), close parenthesis ()), comma (,), and decimal point (.), have specific application in the syntactical expression of the **FORTRAN** statement. The following sections of this manual qualify their use in particular statements and expressions.

In addition to the **FORTRAN** character set, the Varian **FORTRAN IV** system accepts the following characters in Hollerith fields:

"	quotation mark
↑	uparrow
!	exclamation
#	number sign
%	percent
&	ampersand
'	apostrophe
:	colon
;	semicolon

1.3 LINE FORMAT

A **FORTRAN** program consists of a series of statements divided into physical sections called lines that must be coded to a precise grammatical format. **FORTRAN** statements fall into two broad classes, executable and nonexecutable. Executable statements specify program action; nonexecutable statements describe the use of the program, the characteristics of the operands, editing information, statement functions, or data arrangement. The statements of a **FORTRAN** source program are normally written on a standard **FORTRAN** coding form.

Figure 1-1 is a sample **FORTRAN** coding form. The coding form includes 80 columns of information. Columns 73 through 80 are reserved for sequencing information, and have no effect upon the generated execution program. Columns 1 through 72 contain line information in the format described below.



SECTION 1

INTRODUCTION

PROGRAM

PROGRAMMER

DATE

MATRIX MULTIPLICATION

PAGE NO

OF

IDENTIFICATION

M.A.T. NO. 11

FORTRAN STATEMENT

Statement Label

1000

C FOR COMMENT

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69

70

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

65

66

67

68

69</

VT11-0769

Figure 1-1. Sample FORTRAN Coding Form



SECTION 1 INTRODUCTION

1.3.1 Initial Line

The first line of each statement is called an initial line. A statement line consists of three fields: statement number field, continuation flag, and statement field. A statement can include an initial line and continuation lines. Statements can have up to 19 continuation lines as required subject to the following restrictions: DO statements must be wholly contained on an initial line; and the equals character (=) of a replacement statement or a statement function definition must appear on the initial line. An initial line can contain a statement label in columns 1 through 5. In this case, column 6 must contain a zero digit, blank, or space character; and columns 7 through 72 may contain all or part of a statement except for the restrictions noted.

Example

1	5	6	7	10	15	20	25	30	35
			A = .5	*	C	*	D		

1.3.2 Statement Number

Numbers permit statements to be referenced by other portions of a program. A statement number is an integer value in the range 1 to 99999 (leading zeros or blanks are not significant). The initial line of each statement may be given a unique number in columns 1 through 5. The same number cannot be given to more than one statement in a program unit.

Example

1	5	6	7	10	15	20	25	30	35
	5,0		A = .5	*	C +	D			
	6,0		A = .5	*	C +	D			
	8,7,9		A = .5	*	C +	D			



SECTION 1 INTRODUCTION

1.3.3 Continuation Line

Continuation lines are used when additional lines of coding are required to complete a statement originating on an initial line. There can be up to 19 continuation lines per statement with the exceptions previously noted for initial lines. In a continuation line, columns 1 through 5 are blank. Column 6 contains any character other than a zero, blank, or space. The continuation of the statement is in columns 7 through 72. Continuation lines follow an initial line or another continuation line only.

Example

1	5	6	7	10	15	20	25	30	35
			A	=					
		1	.	5	*				
		2	C	+					
		3	D						

1.3.4 Comment Line

Any line with the character C in column 1 is identified as a comment line. Comments can appear anywhere in a program. All comment lines are ignored by a **FORTRAN** compiler, except for display purposes. Comments are in columns 2 through 72.

Example

1	5	6	7	10	15	20	25	30	35
C			T	H	I	S	I	S	A
			C	O	M	M	E	N	T
			S						
			L	I	N	E			



SECTION 1 INTRODUCTION

1.3.5 End Line

Any line containing the character blank in columns 1 through 6 and having only the character string **END** in columns 7 through 72, preceded by, interspersed with, or followed by blank characters, is recognized by the processor as an end line to inform the processor that it has reached the physical end of the program.

Example

1	5	6	7	10	15	20	25	30	35
			END						



SECTION 2 DATA

2.1 GENERAL

Quantities exist as constants or variables. These are distinguished in **FORTRAN** to identify the nature and characteristics of the values encountered in program execution. A constant is a quantity whose value is explicitly stated. A variable is a numeric quantity referenced by name, rather than by its explicit appearance in a program statement. During the execution of a program, a variable can assume many different values.

2.2 DATA TYPES

The Varian **FORTRAN IV** compiler recognizes the following types of data: integer, real (including double precision and complex), logical and Hollerith. Integer data are precise representations of integral values. Real data are approximations of real numbers. Both integer and real data may assume positive, negative, or zero values as follows (zero is considered neither positive nor negative):

	Range in 16-Bit Computers	Range in 18-Bit Computers
Integer	$\pm 32,767$	$\pm 131,071$
Real	approx. $10^{\pm 38}$	approx. $10^{\pm 38}$

2.3 DATA NAMES

FORTRAN data (constants, variables, arrays, and array elements) are identified by names made up of letter or digit strings of one to six characters, the first character of which is a letter. (The character \$ is processed exactly like a letter, but it is reserved for Varian system names. To avoid conflict, therefore, it is advisable not to use the \$ character in names.) Data so identified are implicitly specified as being of type integer or real by the first character, although this can be changed by an explicit specification using a *TYPE* statement. Names beginning with the letters I, J, K, L, M, and N denote integers and other names denote real values.

Examples of integer names are:

I I2A MZXF N5

Examples of real-number names are:

A B2 F5M79 AAA



SECTION 2 DATA

2.4 VARIABLES

Variables are data whose values are derived and defined during program execution. They are identified by symbolic names of the appropriate type.

2.5 CONSTANTS

Constant data are identified explicitly by giving their actual values. Constants do not change in value during program execution. They are specified as integer, real, double precision, complex, Hollerith, or logical constants.

2.5.1 Integer Constants

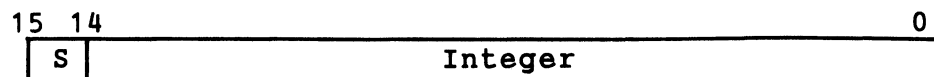
An integer constant is from one to five decimal digits without a decimal point. It can be preceded by a plus (+) or minus (-) sign.

Examples

-217 -32767 +00327 512

In memory, an integer is stored in the format:

16-Bit Computers



18-Bit Computers



2.5.2 Real Constants

A real single-precision constant has one to seven significant digits in any one of the following forms:

$\pm i.$ $\pm i.f$ $\pm .fE\pm e$
 $\pm iE\pm e$ $\pm .f$ $\pm i.E\pm e$
 $\pm i.fE\pm e$



SECTION 2 DATA

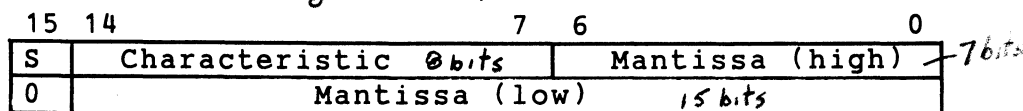
where i, f, and e are each a string of decimal digits representing an integer, fraction and exponent, respectively. The sign character is optional. The decimal point (.) and E characters are obligatory in forms that specify them. If r represents any of the forms preceding $E \pm e$, i.e., $rE \pm e$, then the real constant is interpreted as $(r * 10 \text{ to the power } \pm e)$.

Examples

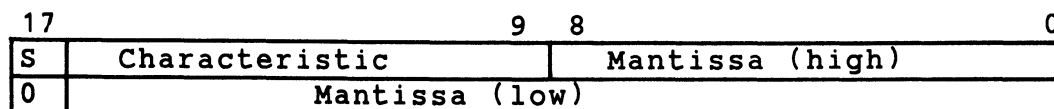
17.	-25.620E-1	0.0
51E1	+.42	-.479
-479E-3	.35E02	

In memory, a real number is stored in the format:

*16-Bit Computers 8 bit charc. + 22 bit Mant.
200₈ bias in charc.*



18-Bit Computers



If a real constant is specified with more significant digits than the precision real data allow, truncation occurs, and only the most significant digits within the range will be represented (seven significant decimal digits for a single-precision real constant and 14 significant decimal digits for a double-precision real constant).

2.5.2.1 Double-Precision Constants

A double-precision constant in a source statement is specified exactly as an E-format real constant, except that the letter E is replaced by D.

Examples

-3476.2D-4	28.D0	.578D + 3
------------	-------	-----------



SECTION 2

DATA

In memory, a double-precision number is stored in the format:

16-Bit Computers *8 bit charac. + 53 bit mant.*

15	14	8	7	0
0	Zeros			Characteristic 8 bits
S	Mantissa (high)			15 bits
0	Mantissa (mid)			15 bits
0	Mantissa (low)			15 bits

18-Bit Computers

17	16	8	7	0
0	Zeros			Characteristic
S	Mantissa (high)			
0	Mantissa (mid)			
0	Mantissa (low)			

The exponent is eight bits long with a bias of 0200. If the mantissa is negative, the high-order word of the mantissa is in one's complement form. All four words are zero for the number zero.

2.5.2.2 Complex Constants

A complex constant is formed by an ordered pair of signed or unsigned real single-precision constants separated by a comma and enclosed in parentheses.

The real constants in a complex constant can be positive, zero, or negative (if unsigned, they are assumed to be positive). The first real constant in a complex constant represents the real part of the complex number; the second represents the imaginary part of the complex number. If the exponent is given, no decimal point is required.

Examples

Valid Complex Constants

(3.2,-1.86)	has the value 3.2-1.86i
(-5.0E + 03,.16E + 02)	has the value -5000. + 16.0i
(4.0E + 03,.16E + 02)	has the value 4000. + 16.0i
(2.1,0.0)	has the value 2.1 + 0.0i

where i equals the square root of -1.



SECTION 2 DATA

Invalid Complex Constants

(292704,1.697)	the real part does not contain a decimal point
(1.2E113,279.3)	the real part contains an invalid decimal exponent
(.003D4,.005D6)	double-precision constants are invalid

In memory, a complex number is stored in the format:

16-Bit Computers

	15		7	6		0
Real	S	Characteristic			Mantissa (high)	
Part	0				Mantissa (low)	
Imaginary	S	Characteristic			Mantissa (high)	
Part	0				Mantissa (low)	

18-Bit Computers

	17	16		9	8		0
Real	S	Characteristic				Mantissa (high)	
Part	0					Mantissa (low)	
Imaginary	S	Characteristic				Mantissa (high)	
Part	0					Mantissa (low)	

2.5.3 Hollerith Constants

A Hollerith constant has the form: nHcccc... where n is the number of characters in the constant and c is a character in the **FORTRAN** character set (section 1.2). It can be used only in the data initialization statement and in the argument list of a **CALL** statement.



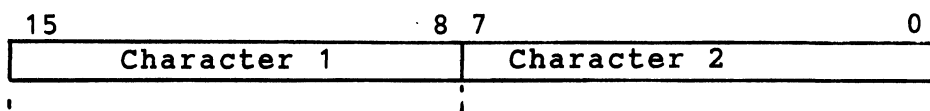
SECTION 2 DATA

Examples

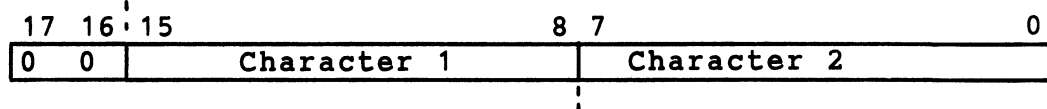
4HABCD
9HTEST CASE

In memory, a Hollerith constant is stored in the format:

16-Bit Computers



18-Bit Computers



For Hollerith constants containing an odd number of characters, the last word contains the last character of the constant as Character 1 and zeros as Character 2.

2.5.4 Logical Constants

A logical constant specifies the logical value of a variable. There are two logical values: `.TRUE.` and `.FALSE.`. Each must be preceded and followed by a period as shown. The logical constants `.TRUE.` and `.FALSE.` specify that the value of the logical variable with which they are associated is true or false, respectively (section 4.4).

In memory, a logical constant is stored in the format:

16-Bit Computers



18-Bit Computers



where `.TRUE.` is stored as zero and `.FALSE.` is stored as minus one.



SECTION 2 DATA

2.6 ARRAYS

An array is an ordered set of data in one, two, or three dimensions identified by a symbolic name. An array declarator (see *DIMENSION* statement) defines the name and size of the array. An array name serves to identify all of the elements in the array, including data type, real or integer.

2.6.1 Array Element

An array element is one member of an array and is identified by a subscript appended to the array name.

2.6.2 Subscripts

A subscript follows the array name and contains one, two, or three subscript expressions enclosed in parentheses. The number of subscript expressions (except in *EQUIVALENCE* statements) corresponds to the specified dimensionality of the array. Two or three expressions within the parentheses must be separated by a comma. Subscript expressions are type integer in one of the following forms:

$$c * v \pm k \quad c * v \quad v \pm k \quad v \quad k$$

where c and k are integer constants and v is an integer variable.

Examples

$$X(2 * J - 3) \quad A(I, J, K) \quad B(20) \quad C(L - 2)$$

2.6.3 Dimensionality

Arrays are stored column-wise in ascending memory locations. Therefore, a two-dimensional real array, A , with three rows and three columns, would be stored internally in the computer as follows:

**SECTION 2**
DATA

Location	Element
$L + 0 \ \& \ L + 1$	$A(1,1)$
$L + 2 \ \& \ L + 3$	$A(2,1)$
$L + 4 \ \& \ L + 5$	$A(3,1)$
$L + 6 \ \& \ L + 7$	$A(1,2)$
$L + 8 \ \& \ L + 9$	$A(2,2)$
.	.
.	.
.	.
$L + 16 \ \& \ L + 17$	$A(3,3)$

The position of an array element, $A(i,j)$, is derived from the following formula:

$$A^1 + (i-1 + 1 \times (j-1)) \times 2$$

where A^1 is the location of the first element in the array, i and j are the specified row and column subscript expressions, and 1 is the number of row elements defined in the array declaration for A , and 2 is the number of words assigned to each real item. In the preceding example, the position of the $A(2,2)$ element would be solved in the following form:

$$L + 0 + (2-1 + 3*(2-1)) \times 2 = L + 8$$



SECTION 3 SPECIFICATIONS AND STATEMENTS

3.1 GENERAL

Specification statements organize and classify data that will be referred to by other statements in the **FORTRAN** program. Specification statements include:

<i>DIMENSION</i>	Names and declares the size of an array.
<i>COMMON</i>	Assigns variable and/or named arrays to common storage areas.
<i>EQUIVALENCE</i>	Assigns variables and named arrays to shared storage areas.
<i>EXTERNAL</i>	Names an external procedure
<i>TYPE</i>	Declares entities to be of type integer, real, double precision, complex, or logical.

Specification statements must precede all other statements except *BLOCK DATA*, *FUNCTION*, *SUBROUTINE*, and *NAME*.

3.2 DIMENSION STATEMENT

Form: *DIMENSION* V1(i1), V2(i2), ..., Vn(in), where each V(i), (called an array declarator), is composed of a declarator name V (the name of the array), and a declarator subscript (i). Each (i) is an unsigned integer constant, two unsigned integer constants separated by a comma, or three unsigned integer constants separated by commas. Each constant must have a value greater than zero and less than the limit of available memory.

A *DIMENSION* statement specifies that the declarator names listed are arrays in the program unit. The number of dimensions and the maximum size of each dimension is specified by the declarator subscript associated with each declarator name.

More than one *DIMENSION* statement can appear in a program.

An array element is referred to by the array name qualified by a subscript to identify the desired element. If the value of this subscript is out of the range specified by the array declarator, the derived computational results will be unpredictable.



SECTION 3 SPECIFICATIONS AND STATEMENTS

Array elements are stored column-wise in computer memory from low to high address storage. Therefore, one-dimension arrays are stored sequentially in the order $A(1)$, $A(2)$, ..., $A(n)$, while two-dimension arrays are stored with the first (leftmost) dimension varying most rapidly, i.e., $A(1,1)$, $A(2,1)$, ..., $A(m,1)$, $A(1,2)$, $A(2,2)$, ..., $A(m,n)$.

Example

```
DIMENSION A(5), I1(3,6), C(5,10), BIG(10,10,10)
```

This specification statement indicates that A is a real vector with five elements; $I1$ is an integer matrix of size $3 \times 6 = 18$ elements; C is a real matrix of size $5 \times 10 = 50$ elements; and BIG is a real matrix of size $10 \times 10 \times 10 = 1000$ elements.

3.3 COMMON STATEMENT

General Form

```
COMMON /x/a,b,.../r/c,d,...
```

where:

$a,b,...,c,d,...$ are variable or array names or array declarators.

$/x/.../r/$ represents optional common block names consisting of one through six alphanumeric characters, the first of which is alphabetic. These names must always be embedded in slashes.

Although the **COMMON** statement may be used to provide dimension information, adjustable dimensions may never be used.

Variables or arrays that appear in a calling program or subprogram may be made to share the same storage locations with variables or arrays in other subprograms by use of the **COMMON** statement. For example, if one program contains the statement:

COMMON TABLE

as its first **COMMON** statement, and a second program contains the statement:



SECTION 3 SPECIFICATIONS AND STATEMENTS

COMMON TREE

as its first *COMMON* statement and the two programs are loaded together, the variable names *TABLE* and *TREE* refer to the same storage location.

If the main program contains the statement:

```
COMMON A, B, C
```

and a subprogram contains the statement:

```
COMMON X, Y, Z
```

then *A* shares the same storage location as *X*, *B* shares the same storage location as *Y*, and *C* shares the same storage location as *Z*.

Common entries appearing in *COMMON* statements are cumulative in the given order throughout the program; that is, they are cumulative in the sequence in which they appear in all *COMMON* statements. For example, consider the following two *COMMON* statements:

```
COMMON A, B, C  
COMMON G, H
```

These two statements have the same effect as the single statement:

```
COMMON A, B, C, G, H
```

Redundant entries are not allowed. For example, the following statement is invalid:

```
COMMON A, B, C, A
```

Consider the following example:

Example

CALLING PROGRAM

·
·

SUBPROGRAM SUBROUTINE

MAPMY(...)

·



SECTION 3 SPECIFICATIONS AND STATEMENTS

Example

```
DIMENSION A(5), I1(3,3), B1(3)
COMMON B, B1, B2
EQUIVALENCE (X, A(2), Y), (B, C2, F5), (I1(5), B2)
```

The effect of an *EQUIVALENCE* statement upon *COMMON* assignments may be the lengthening of *COMMON*. This lengthening is permitted only if it increases *COMMON* in the same direction as additional *COMMON* elements would. Thus, in this example, the equivalence (B1(1), A(3)) would have been invalid. It is also invalid to equate two elements of the same array to each other.

3.5 TYPE STATEMENT

General Form:

```
TYPE a, b, ..., z
```

where:

TYPE is *INTEGER*, *REAL*, *DOUBLE PRECISION*, *COMPLEX*, or *LOGICAL*

a, b, ..., z represent variable or array names, or array declarators

The *TYPE* statements declare the type *INTEGER*, *REAL*, *DOUBLE PRECISION*, *COMPLEX* or *LOGICAL* of a particular variable or array by its name, rather than by its initial character. This differs from the other way of specifying the type of a variable or array (i.e., implicit type specification).

Example 1

```
INTEGER ITEM, VALUE
```

Explanation:

This statement declares that the variables *ITEM* and *VALUE* are of type *INTEGER*.



SECTION 3 SPECIFICATIONS AND STATEMENTS

Example 2

REAL ARRAY HOLD, VALUE, ITEM (5, 5)

Explanation:

This statement declares that the variables ARRAY, HOLD, VALUE, and the array named item are of type *REAL*. In addition, it declares the size of the array ITEM. The variables ARRAY, HOLD, and VALUE have two storage locations reserved for each; and the array named ITEM has 50 storage locations reserved (two for each variable in the array).

Example 3:

DOUBLE PRECISION C, D, E

Explanation:

This statement declares that the variables C, D, and E are of type *DOUBLE PRECISION*. Thus, C, D, and E each have four storage locations reserved, one for the exponent and three for the mantissa (section 2.5.3).

Example 4

COMPLEX C, D, E

Explanation:

This statement declares that the variables C, D, and E are of type *COMPLEX*. Thus C, D, and E each have four storage locations reserved (two for the real part, two for the imaginary part).



SECTION 4 EXPRESSIONS AND ASSIGNMENTS

4.1 ARITHMETIC EXPRESSIONS

An arithmetic expression is formed in **FORTRAN** syntax by a combination of operators and elements. The expression and its elements identify the expression to be type *INTEGER*, *REAL*, *DOUBLE PRECISION*, or *COMPLEX*.

The arithmetic operators are:

Operator	Representing
+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

The arithmetic elements are described by the following statements:

<i>PRIMARY</i>	An <i>ARITHMETIC EXPRESSION</i> enclosed in parentheses, a constant, a variable reference, an array element reference, or function reference.
<i>FACTOR</i>	A <i>FACTOR</i> is <i>PRIMARY</i> or <i>PRIMARY**PRIMARY</i>
<i>TERM</i>	A <i>TERM</i> is a <i>FACTOR</i> or one of the forms: <i>TERM/FACTOR</i> <i>TERM*TERM</i>
<i>SIGNED TERM</i>	A <i>TERM</i> immediately preceded by a + or - sign.
<i>SIMPLE ARITHMETIC EXPRESSION</i>	A <i>TERM</i> or two <i>SIMPLE ARITHMETIC EXPRESSIONS</i> separated by a + or -sign.
<i>ARITHMETIC EXPRESSION</i>	A <i>SIMPLE ARITHMETIC EXPRESSION</i> or a signed <i>TERM</i> or either of the preceding



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

immediately followed by a + or -sign and
a *SIMPLE ARITHMETIC EXPRESSION*.

A *PRIMARY* of any type may be exponentiated by an *INTEGER PRIMARY* and the resulting factor is of the same type as that of the element being exponentiated. A *REAL* or *DOUBLE PRECISION PRIMARY* may be exponentiated by a *REAL* or *DOUBLE PRECISION PRIMARY*. The resultant *FACTOR* is of type *REAL* if both *PRIMARIES* are *REAL*, and otherwise of type *DOUBLE PRECISION*. These are the only cases for which use of the exponentiation operator is defined. Valid combinations for exponentiation are:

Base	Exponent
<i>REAL</i>	<i>REAL, INTEGER or DOUBLE PRECISION</i>
<i>INTEGER</i>	<i>INTEGER (REAL and DOUBLE PRECISION exponents are invalid)</i>
<i>DOUBLE PRECISION</i>	<i>REAL, INTEGER or DOUBLE PRECISION</i>

By use of the arithmetic operators other than exponentiation, any admissible element may be combined with another admissible element of the same type, and the resultant element is of the same type.

Further, an admissible real element may be combined with an admissible double-precision or complex element; the resultant element is of type *DOUBLE PRECISION* or *COMPLEX*, respectively.

A part of an expression is evaluated only if it is necessary to establish the value of the expression. The rules for formation of expressions imply the binding strength of operators. The range of the subtraction operator is the term of the operator that immediately succeeds it. The evaluation may proceed according to any valid formation sequence. Use of an array element name requires the evaluation of its subscript. The type of the expression in which a function reference or subscript appears does not affect, nor is it affected by, the evaluation of the actual arguments or subscript. An element whose value is not mathematically defined cannot be evaluated.

The following rules represent the derivation of all permissible expressions:

A variable, constant, or function standing alone is an expression.



SECTION 4 EXPRESSIONS AND ASSIGNMENTS

A(1)
JOBNO
217
17.26
SQRT(A + B)

If E is an expression whose first character is not an operator, then +E and -E are expressions.

-A(1)
+ JOBNO
-217
+ 17.26
-SQRT(A + B)

If E is an expression, then (E) is an expression meaning the quantity E taken as a unit.

(-A)
-(+ JOBNO)
-(X + Y)
(A-SQRT(A + B))

If E is an expression whose first character is not an operator, and F is an expression, then: F + E, F-E, F*E, F/E and F**E are all expressions.

-(B(I,J) + SQRT(A + B(K,L)))
-(B(I + E, 3*J + K) + A)
1.7E-2**(X + 5.0)

The mode of expression is determined by the modes of its elements, which must be the same with the following exceptions:

- a. A *REAL* quantity can appear in an *INTEGER* expression only as an argument of a function.
I + LFUNC(B)
- b. An *INTEGER* quantity can appear in a *REAL* expression only as an argument of a function, or as a subscript or an exponent.
AFUNC(I + 2)
A(I, J + 1)
B**N



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

The order of evaluation of expressions is established by the use of parentheses in the statement. If parentheses are not indicated, the following conventions of mathematics apply.

The hierarchy of operations, in order of precedence is: exponentiation, followed by multiplication and division, followed by addition and subtraction.

Within the same hierarchy of operations, evaluation proceeds from left to right.

Examples

$X + Y * Z$	is interpreted as	$X + (Y * Z)$
$W * X / Y * Z$	is interpreted as	$((W * X) / Y) * Z$
$B ** 2 - 4 * A * C$	is interpreted as	$(B ** 2) - ((4 * A) * C)$
$X - Y - Z$	is interpreted as	$(X - Y) - Z$
$X / Y / Z$	is interpreted as	$(X / Y) / Z$
$-X ** 3$	is interpreted as	$-(X ** 3)$

4.2 ARITHMETIC ASSIGNMENT STATEMENT

General Form

$$a = b$$

where:

a is any subscripted or nonsubscripted variable.

b is any arithmetic expression.

This **FORTRAN** statement closely resembles a conventional algebraic equation; however, the equal sign specifies replacement rather than equivalence. That is, the expression to the right of the equal sign is evaluated, and the resulting value replaces the current value of the variable to the left of the equal sign.



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

Rules for Assignment of b to a

If a is TYPE	and b is TYPE	the assignment rule is*
INTEGER	INTEGER	Assign
INTEGER	REAL	Fix and Assign
INTEGER	DOUBLE PRECISION	Fix and Assign
INTEGER	COMPLEX	P
REAL	INTEGER	Float and Assign
REAL	REAL	Assign
REAL	DOUBLE PRECISION	DP Evaluate and Real Assign
REAL	COMPLEX	P
DOUBLE PRECISION	INTEGER	DP Float and Assign
DOUBLE PRECISION	REAL	DP Evaluate and Assign
DOUBLE PRECISION	DOUBLE PRECISION	Assign
DOUBLE PRECISION	COMPLEX	P
COMPLEX	INTEGER	P
COMPLEX	REAL	P
COMPLEX	DOUBLE PRECISION	P
COMPLEX	COMPLEX	Assign

*Notes

P =	prohibited combination
Assign =	transmit the resulting value without change
Real assign =	transmit as much precision of the most significant part of the resulting value as a <i>REAL</i> datum can contain
DP evaluate =	evaluate according to the most precise rules, then DP float
Fix =	truncate any fractional part and transform to <i>INTEGER</i>
Float =	transform to <i>REAL</i>
DP float =	transform to <i>DOUBLE PRECISION</i> retaining as much precision as a <i>DOUBLE PRECISION</i> datum can contain



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

Assume that the type of the following variables has been specified as:

Variable Names	Type
I, J, W	<i>INTEGER</i> variables
A, B, C, D	<i>REAL</i> variables
E	<i>COMPLEX</i> variable

Then the following examples illustrate valid arithmetic statements using constants, variables, and subscripted variables of different types:

Statement	Description
$A = B$	The value of A is replaced by the current value of B.
$W = B$	The value of B is truncated to an integer value, and this value replaces the value of W.
$A = I$	The value of I is converted to a real value, and this result replaces the value of A.
$I = I + 1$	The value of I is replaced by the value of $I + 1$.
$A = C * D$	The most significant part of the product of C and D replaces the value of A.
$E = (1.0, 2.0)$	The value of the complex variable E is replaced by the complex constant (1.0, 2.0). Note that the statement $E = (A, B)$ where A and B are real variables, is invalid.

4.3 LOGICAL ASSIGNMENT STATEMENT

General Form

$$a = b$$



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

where:

a is a subscripted or nonsubscripted variable.

b is any logical expression.

Variable Names	Type
G, H	LOGICAL variables

Examples of logical assignment statements are:

Statement	Description
G = .TRUE.	The value of G is replaced by the logical constant .TRUE..
H = .NOT.G	If G is .TRUE., the value of H is replaced by the logical constant .FALSE.. If G is .FALSE., the value of H is replaced by the logical constant .TRUE..
G = 3..GT.I	The value of I is converted to a real value; if the real constant 3. is greater than this result, the logical constant .TRUE. replaces value of G. If 3. is not greater than I, the logical constant .FALSE. replaces the value of G.

4.4 LOGICAL EXPRESSIONS

The simplest form of logical expression consists of a single logical constant, logical variable, or logical subscripted variable, the value of which is always a truth value (i.e., either .TRUE. or .FALSE.).

More complicated logical expressions may be formed by using logical and relational operators. These expressions may be in one of the three following forms:

- a. Relational operators combined with arithmetic expressions whose mode is *INTEGER*, *REAL*, or *DOUBLE PRECISION*.



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

- b. Logical operators combined with logical constants (.TRUE. and .FALSE.), logical variables, or subscripted logical variables.
- c. Logical operators combined with either or both forms of the logical expressions described in items a and b.

Item a is discussed in the following section, Relational Operators; items b and c are discussed in the section entitled Logical Operators.

4.4.1 Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator and has the value .TRUE. or .FALSE. as the relation is true or false. Both arithmetic expressions can be type *INTEGER* or one may be type *REAL* or *DOUBLE PRECISION* and the other type *REAL* or *DOUBLE PRECISION*. If a real and a double precision expression appear in a relational expression, the effect is the same as a similar relational expression. This similar expression contains a double precision zero as the right-hand arithmetic expression and the difference of the two original expressions (in their original order) as the left. The relational operator is unchanged.

The six relational operators, each of which must be preceded and followed by a period, are as follows:

Relational Operator	Definition
.GT.	Greater than ($>$)
.GE.	Greater than or equal to ($\geq$)
.LT.	Less than ($<$)
.LE.	Less than or equal to ($\leq$)
.EQ.	Equal to ($=$)
.NE.	Not equal to ($\neq$)

The relational operators express an arithmetic condition which can be either true or false. Only arithmetic expressions whose mode is *INTEGER*, *REAL*, or *DOUBLE PRECISION* can be combined by relational operators. For example, assuming the type of variable has been specified as follows:



SECTION 4 EXPRESSIONS AND ASSIGNMENTS

Variable Names	Type
ROOT, E, Q	REAL variables
A, I, F	INTEGER variables
L	LOGICAL variable
C	COMPLEX variable

then, the following illustrates valid and invalid logical expressions using the relational operators.

Example

Valid Logical Expressions Using Relational Operators:

(ROOT*Q).GT.E

A.LT.I

E**2.7.EQ.(5.*ROOT + 4.)

57.9.LE.(4.7 + E)

.5.GE.9.*ROOT

E.EQ.27.3E + 05

Invalid Logical Expressions Using Relational Operators:

C.LT.ROOT Complex quantities can never appear in logical expressions.

C.GE.(2.7,5.9E3) Complex quantities can never appear in logical expressions.

L.EQ.(A + F) Logical quantities may never be joined by relational operators.

E**2.EQ97.1E9 Missing period immediately after the relational operator.

.GT.9 Missing arithmetic expression before the relational operator.



SECTION 4

EXPRESSIONS AND ASSIGNMENTS

4.4.2 Logical Operators

The three logical operators, each of which must be preceded and followed by a period, are as follows: (A and B represent logical constants or variables, or expressions containing relational operators).

Logical Operator	Definition
.NOT.	.NOT.A – if A is .TRUE., then .NOT.A has the value .FALSE.; if A is .FALSE., then .NOT.A has the value .TRUE.
.AND.	A.AND.B – if A and B are both .TRUE., then A.AND.B has the value .TRUE.; if either A or B or both are .FALSE., then A.AND.B has the value .FALSE.
.OR.	A.OR.B – if either A or B – or both are .TRUE., then A.OR.B has the value .TRUE.; if both A and B are .FALSE., then A.OR.B has the value .FALSE.

Two logical operators may appear in sequence only if the second one is the logical operator .NOT..

Only those expressions which, when evaluated, have the value .TRUE. or .FALSE. may be combined with the logical operators to form logical expressions. For example, assume that the type of variable has been specified as follows:

Variable Names	Type
ROOT, E, Q	REAL variables
A, I, F	INTEGER variables
L, W	LOGICAL variables
C	COMPLEX variable



SECTION 4 EXPRESSIONS AND ASSIGNMENTS

Then the following examples illustrate valid and invalid logical expressions using both logical and relational operators.

Examples

Valid Logical Expressions:

(ROOT*Q.GT.E).AND.W

L.AND..NOT.(I.GT.F)

(E + 5.9E2.GT.2.*E).OR.L

.NOT.W.AND..NOT.L

L.AND..NOT.W.OR.I.GT.F

(E**F.GT.ROOT).AND..NOT.(I.EQ.A)

Invalid Logical Expressions

A.AND.L

A is not a logical expression.

.OR.W

.OR. must be preceded by a logical expression.

NOT.(A.GT.F)

missing period before the logical operator
.NOT.

(C.EQ.I).AND.L

a complex variable may never appear in a
logical expression.

L.AND..OR.W

the logical operators .AND. and .OR. must
always be separated by a logical expression.

.AND.L

.AND. must be preceded by a logical expres-
sion.



SECTION 4 EXPRESSIONS AND ASSIGNMENTS

Order of Computations in Logical Expressions

Where parentheses are omitted, or where the entire logical expression is enclosed within a single pair of parentheses, the order in which the operations are performed is as follows:

Operation	Hierarchy
Evaluation of Functions	1st (highest)
Exponentiation (**)	2nd
Multiplication and division (* and /)	3rd
Addition and subtraction (+ and -)	4th
.LT.,.LE.,.EQ.,.NE.,.GT.,.GE.	5th
.NOT.	6th
.AND.	7th
.OR.	8th

For example, the expression:

`(A.GT.D**B.AND..NOT.L.OR.N)`

is effectively evaluated in the following order.

- | | |
|------------|---|
| 1. D**B | Call the result W (exponentiation) |
| 2. A.GT.W | Call the result X (relational operator) |
| 3. .NOT.L | Call the result Y (highest logical operator) |
| 4. X.AND.Y | Call the result Z (second highest logical operator) |
| 5. Z.OR.N | Final operation |

Use of Parentheses in Logical Expressions

Parentheses may be used in logical expressions to specify the order in which the operations are to be performed. Where parentheses are used, the expression contained within the most deeply nested parentheses (that is, the innermost pair of parentheses) is effectively evaluated first. For example, the logical expression:

`((I.GT.(B + C)).AND.L)`

is effectively evaluated in the following order.



SECTION 4 EXPRESSIONS AND ASSIGNMENTS

- | | |
|------------|-------------------|
| 1. B + C | Call the result X |
| 2. I.GT.X | Call the result Y |
| 3. Y.AND.L | Final operation |

The logical expression to which the logical operator `.NOT.` applies must be enclosed in parentheses if it contains two or more quantities. For example, assume that the values of the logical variables, A and B, are `.FALSE.` and `.TRUE.`, respectively. Then the following two expressions are not equivalent:

`.NOT.(A.OR.B)`
`.NOT.A.OR.B`

In the first expression, `A.OR.B` is evaluated first. The result is `.TRUE.`, but `.NOT. (.TRUE.)` implies `.FALSE.`. Therefore, the value of the first expression is `.FALSE.`.

In the second expression, `.NOT.A` is evaluated first. The result is `.TRUE.`; but `.TRUE.OR.B` implies `.TRUE.`. Therefore, the value of the second expression is `.TRUE.`.



SECTION 5 CONTROL STATEMENTS

5.1 GENERAL

Each statement in a **FORTRAN** program is executed in the order of its appearance in the source program, unless this sequence is interrupted or modified by a control statement. This section of the manual describes the control statements used in **FORTRAN**.

5.2 GO TO STATEMENTS

GO TO statements transfer logical control from one section of a program to another. **FORTRAN** includes three forms of the *GO TO* statement: unconditional, computed, and assigned *GO TO*.

5.2.1 Unconditional GO TO

An unconditional *GO TO* is of the form: *GO TO k*, where *k* is a statement label reference.

Execution of this statement causes the statement identified by the label *k* to be executed next in sequence.

Example

```
GO TO 72
.
71 V7 = HQ(5) + Y**L
.
.
.
72 V7 = HQ(4) + X**J
```

In this example, execution of the *GO TO 72* statement causes statement number 71 and any succeeding statements to be bypassed. Execution is resumed with statement number 72.

5.2.2 Computed GO TO

The computed *GO TO* statement is of the form: *GO TO (k1, k2, ..., kn), i*, where the *k*'s are statement label references, and *i* is an integer variable reference.

Execution of this statement causes the statement identified by the statement label *k_j* to be executed next in sequence, where *j* is the value of *i* at execution time. Valid execution of



SECTION 5 CONTROL STATEMENTS

this statement is dependent upon the value of the integer variable such that 1 is less than or equal to j, and j is less than or equal to n.

Example

GO TO (98, 405, 3), n

Execution of the statement in the example will cause control to be transferred to the statement labeled 98, 405, or 3 if the value of the variable integer n is 1, 2, or 3, respectively.

5.2.3 ASSIGN and Assigned GO TO

General Form

```
ASSIGN i TO m
      .
      .
      .
GO TO m, (X1, X2, X3,...,Xn)
```

where:

i is an executable statement number.

X1,X2,X3,...,Xn are executable statement numbers.

m is a nonsubscripted integer variable that is assigned one of the following statement numbers: X1, X2, X3,...Xn.

The assigned GO TO statement causes control to be transferred to the statement numbered X1, X2, X3,..., or Xn, depending on whether the current assignment of m is X1, X2, X3,..., or Xn, respectively. For example, in the following statement:

GO TO N, (10, 25, 8)

If the current assignment of the integer variable N is statement number 8, then the statement numbered 8 is executed next. If the current assignment of N is statement number 10, the statement numbered 10 is executed next. If N is assigned statement number 25, statement 25 is executed next.



SECTION 5 CONTROL STATEMENTS

The current assignment of the integer variable *m* is determined by the last executed *ASSIGN* statement. Only an *ASSIGN* statement may be used to initialize or change the value of *m*. The value of *m* is not the integer statement number; *ASSIGN 10 TO I* is not the same as *I = 10*.

Example 1

```
.  
.   
.   
ASSIGN 50 TO NUMBER  
.   
10 GO TO NUMBER, (35, 50, 25, 12, 18)  
.   
.   
.   
50 A = B + C  
.   
.   
. 
```

In the above example, statement 50 is executed immediately after statement 10.

Example 2

```
.   
.   
.   
ASSIGN 10 TO ITEM  
.   
.   
.   
13 GO TO ITEM, (8, 12, 25, 50, 10)  
.   
.   
.   
8 A = B + C  
.   
. 
```

(continued)



SECTION 5

CONTROL STATEMENTS

In statement 5, if the value of the expression is `.TRUE.` (i.e., A is less than or equal to 0.0), the statement `GO TO 25` is executed next and control is passed to the statement numbered 25. If the value of the expression is `.FALSE.` (i.e., A is greater than 0.0), the statement `GO TO 25` is ignored and control is passed to the statement numbered 10.

In statement 15, if the value of the expression is `.TRUE.` (i.e., A is equal to B), the value of `ANSWER` is replaced by the value of the expression $(2.0 * A / C)$, and statement 20 is executed. If the value of the expression is `.FALSE.` (i.e., A is not equal to B), the value of `ANSWER` remains unchanged and statement 20 is executed next.

Example

```
.  
.   
.   
5 IF (P.OR..NOT.Q) A = B  
10 C = B**2  
.   
.   
.
```

Assume that P and Q are logical variables. In statement 5, if the value of the expression is `TRUE.`, the value of A is replaced by the value of B and statement 10 is executed next. If the value of the expression is `.FALSE.`, statement `A = B` is skipped and statement 10 is executed.

5.4 CALL STATEMENT

The `CALL` statement causes a transfer of execution control to a subroutine-type subprogram, and is of one of the forms: `CALL s(a1, a2,..., an)` and `CALL s`, where s is the name of a subroutine and the a's are actual arguments that will replace the dummy arguments in the called subroutine. Arguments can be Hollerith constants, variable names, array element names, array names, any other expression, or the name of an external procedure. They must, however (except for Hollerith constants), be indicated in order, number, and type with the corresponding dummy arguments of the subroutine.



SECTION 5 CONTROL STATEMENTS

Execution of the *CALL* statement transfers control to the designated subroutine. The arguments declared in the statement line are associated with the dummy arguments that are parameters of the executable statements of the subroutine. Control is then passed to the first executable statement of the called subroutine. Control will be returned to the first executable statement following the *CALL* statement upon execution of the *RETURN* statement in the subroutine. Examples of calling sequences to subroutines are shown below.

```
CALL TEST (A,I)  
CALL EXIT
```

The first example will transfer execution control to the subroutine labelled *TEST* and include the parameters or arguments *A* and *I* in the subroutine. The second example will cause execution control to be transferred to the subroutine labelled *EXIT*. Any arguments required for execution of exit are self-contained in the logic of the subroutine.

5.5 RETURN STATEMENT

The execution of a *RETURN* statement results in the exit from a subprogram, and is expressed in the form: *RETURN*. A *RETURN* statement defines the logical end of a procedure subprogram and, therefore, may appear only in a subprogram. Execution of the statement returns logical control to the current calling program unit. Each subprogram must contain at least one *RETURN* statement.

In the case of a subroutine subprogram, control is returned to the first statement immediately following the *CALL* statement that released control to the subroutine. In the case of a function subprogram, control is returned (with the value of the function available) to the statement that called the function subprogram.

5.6 CONTINUE STATEMENT

Form: *CONTINUE*.

The *CONTINUE* statement results in no action in an execution sequence; therefore, the statement has no effect upon the program. This statement serves as a program unit reference point and is frequently used at the end of a *DO* loop.

Example

```
IF (I) 10, 11, 12  
10 V7 = HQ(5) + Y**L
```

(continued)



SECTION 5 CONTROL STATEMENTS

```
.  
.
GO TO 13
11 V7 = HQ(4) + X**J
.  
.
GO TO 13
12 V7 = HQ(3) + X**L
.  
.
13 CONTINUE
```

5.7 PAUSE STATEMENT

Form: PAUSE n or PAUSE, where n is an octal digit string of length from one to five digits designating the particular *PAUSE*.

A *PAUSE* statement causes a temporary cessation of program execution and displays PAUSE n (see section 8 for display format). The statement permits operator intervention for setup or control functions, such as changing data tapes. The computer executes a halt instruction, delaying further execution until the operator selects the console RUN button. Execution will resume at the first executable statement following the *PAUSE* statement.

Example

```
PAUSE 01
```

5.8 STOP STATEMENT

Form: STOP n or STOP, where n is an octal digit string of length from one to five digits designating the particular *STOP*.

A *STOP* statement causes termination of program execution and displays STOP n (see section 8 for display format). The program then terminates.

Example

```
STOP 0721
```



SECTION 5 CONTROL STATEMENTS

5.9 DO STATEMENT

The *DO* statement controls repetitive execution of a group of statements. The number of repetitions depends on the value of a control variable. The statement assumes one of the forms: $DO\ n\ i = m1, m2, m3$ and $DO\ n\ i = m1, m2$, where n is the statement label of an executable statement. This statement, called the terminal statement of the associated *DO* must physically follow and be in the same program unit as the *DO* statement. The terminal statement may not be a *GO TO* of any form, arithmetic *IF*, *RETURN*, *STOP*, *PAUSE*, or another *DO* statement, nor a logical *IF* statement containing one of these forms.

Symbol i is an integer variable name, identified as the control variable.

Symbol $m1$, identified as the initial parameter, $m2$, as the terminal parameter, and $m3$, as the incrementation parameter are each either an integer constant or integer variable reference. If the second form of the *DO* statement is used, a value of 1 is implied for the incrementation parameter. When the *DO* statement is executed, the values of $m1$, $m2$, and $m3$ must be greater than zero.

Associated with each *DO* statement is a range that is defined to be those executable statements from and including the first executable statement following the *DO*, to and including the terminal statement defined by the *DO*. A special situation occurs when the range of a *DO* contains another *DO* statement. In this case, the range of the contained *DO* must be a subset of the range of the containing *DO*.

The control variable is assigned the value represented by the initial parameter. This value must be less than or equal to the value represented by the terminal parameter.

The range of the *DO* is executed.

If control reaches the terminal statement after execution of the terminal statement, the control variable of the most recently executed *DO* statement associated with the terminal statement is incremented by the value represented by the associated incrementation parameter.

If the value of the control variable is greater than the value represented by its associated terminal parameter, the *DO* is said to be satisfied, and the control variable becomes undefined.

If there were one or more other *DO* statements referring to the terminal statements in question, the control variable of the next most recently executed *DO* statement is



SECTION 5

CONTROL STATEMENTS

incremented by the value represented by the associated incrementation parameter until all *DO* statements referring to the particular termination statement are satisfied, at which time the first executable statement following the terminal statement is executed.

Upon exiting from the range of a *DO* by execution of a *GO TO* statement or an arithmetic *IF* statement, that is other than by satisfying the *DO*, the control variable of the *DO* is defined and is equal to the most recent attained value.

A *GO TO* or arithmetic *IF* statement may not cause control to pass into the range of a *DO* from outside its range. When a procedure reference occurs in the range of a *DO*, the actions of that procedure are considered to be temporarily within that range, i.e., during the execution of that reference.

The control variable, initial, terminal, and incrementation parameters of a *DO* may not be redefined during the execution of the range of that *DO*.

If a statement is the terminal statement of more than one *DO* statement, the label of that terminal statement may not be used in any *GO TO* or arithmetic *IF* statement that occurs anywhere but in the range of the most deeply contained *DO* with that terminal statement.

Example

```
DO 607 K1=2, ID, 3
```

The foregoing statement would cause K1, the control variable, to be set to the value of the initial parameter, 2. Execution would proceed at the statement immediately following, down to and including the statement identified by the label 607. After each execution of the loop, K1 is incremented by the incrementation parameter, 3, and evaluated in relation to the current value of the terminal parameter, ID. If the current value of K1 = ID, execution control is transferred to the statement following that identified by the label 607; otherwise, the *DO* cycle is repeated.

Example illustrating *DO* nesting:

```
WRITE (MX,8)
L = 0
DO 150 J = 1,K
DO 140 I = 1,M
L = L + 1
140 D(I) = V(L)
```



SECTION 5
CONTROL STATEMENTS

```
150  WRITE (MX,9)J,(D(I),I = 1,M)
      CALL LOAD (M,K,R,V)
C    PRINT FACTOR MATRIX
      WRITE (MX,10)K
      DO 180 I = 1,M
      DO 170 J = 1,K
      L = M*(J-1) + I
170   D(J) = V(L)
180   WRITE (MX,11)I,(D(J),J = 1,K)
      IF (K-1) 185, 185, 188
185   WRITE (MX,19)K
      GO TO 100
188   CALL VARMX (M,K,V,NC,TV,B,T,D)
```




SECTION 6 INPUT/OUTPUT STATEMENTS

6.1 GENERAL

Input statements provide a program with the means of receiving information from external sources. Output statements allow the transmission of program data to external sources. These external sources may be devices such as magnetic tape and paper tape handlers, typewriters, and punch card processors.

There are two types of input/output statements.

READ and *WRITE* statements
AUXILIARY statements

The first statement type causes the transfer of records of sequential files to and from the program. These data may be formatted information consisting of strings of characters or unformatted information consisting of binary word values in the form in which they normally appear in storage. The second statement type consists of the *BACKSPACE* and *REWIND* statements, which provide for positioning of devices and the *ENDFILE* statement, which provides for writing a tape mark.

Input/output statements reference input/output units, formatted information, and format specifications. An input/output unit is identified by a logical unit number, *u*, which can be an integer constant or a variable name referencing an integer constant. Logical unit number assignments for **FORTRAN** are listed in section 8 and appendix A. The format specification is defined by a *FORMAT* statement having the statement label *f*. This statement must appear in the same program as the input/output statement.

6.2 INPUT/OUTPUT LISTS

The input list specifies the names of variables and array elements to which input values are assigned. The output list specifies the names of variables and array elements whose values are to be transmitted. Input and output lists are of the same form.

6.3 SIMPLE LISTS

Simple lists have the form: *m1, m2, m3...., mn*, where *mi* is the name of a variable or array element. Commas separate each name in the list. The period signifies possible additional list items. List elements can be enclosed in parentheses.



SECTION 6

INPUT/OUTPUT STATEMENTS

Example

Input Lists

A
C(26,L)
R, K, D, (I, J)

Output Lists

B
I(10,10)
S, (R, K), F(1,25)

An array variable in a list that is not subscripted is considered equivalent to the listing of each successive element of the array. If B is an array, list B is equivalent to B (1, 1), B (2, 1), B (3, 1),..., B (1, 2), B (2, 2),..., B (j, k), where j and k are the subscript limits of B.

6.4 DO-IMPLIED LISTS

A DO-implied list is a simple list followed by a comma character and an expression of the form: $i = m1, m2, m3$ or $i = m1, m2$.

The elements i , $m1$, $m2$, and $m3$ have the same meaning as defined for the DO statement. The DO implication applies to all simple list items enclosed in parentheses with the implication. For input lists, i , $m1$, $m2$, and $m3$ may appear within this range only as subscripts.

Examples

DO-Implied Lists:

(X (I), I = 1, 4)
(Q (J), R (J), J = 1, 2)
(G (K), K = 1, 7, 3)
((A (I, J), I = 3, 5), J = 1, 2)
(X (K), K = 1, 2), I, (R (J), J = 3, 5)

Equivalent Simple Lists:

X (1), X (2), X (3), X (4)
Q (1), R (1), Q (2), R (2)
G (1), G (4), G (7)
A (3, 1), A (4, 1), A (5, 1)
A (3, 2), A (4, 2), A (5, 2)
X (1), X (2), I, R (3), R (4), R (5)



SECTION 6 INPUT/OUTPUT STATEMENTS

6.5 READ STATEMENTS

These statements are used to obtain data values from an external source. The data values are input in either formatted or unformatted mode. The form of a formatted *READ* statement is:

`READ (u,f) k.`

The verb *READ* and the parentheses must appear in this form.

Execution of this statement causes information to be transmitted from the external source whose logical unit number is defined by *u* (section 8.13 and appendix A). These data are scanned and converted as specified by the format specification, *f*, and the resulting values are assigned to the variable names defined in the list, *k*.

The form of an unformatted *READ* statement is: `READ (u) k.`

The verb *READ* and the parentheses must appear in this form.

This statement causes data to be input in binary form from the unit defined by *u*. The values are assigned to the variable names defined in the list, *k*.

Examples

```
READ (1,44) A, B, C
READ (2) R, S
READ (N, 12) A, (R (I), I = 1, 10)
READ (L) S, (T (J), J = 1, N)
```

All information appearing on external sources is divided into records. Each time a *READ* statement is executed, a new record is processed. The number of records input by a single *READ* statement is determined by the list and format specification. If only part of a record is input the remainder of the record is lost as the next *READ* processes the next record. Records are read sequentially until the list is exhausted. Only enough values are read to fill the list.

The list, *k*, in an unformatted *READ* statement may be left blank to skip a record.

Formatted and unformatted records are described in sections 8.5 and 8.6.



SECTION 6 INPUT/OUTPUT STATEMENTS

6.6 WRITE STATEMENTS

WRITE statements are used to transfer program data to external devices. These data may be formatted or unformatted. The form of a formatted *WRITE* statement is: *WRITE* (u, f) k.

The verb *WRITE* and the parentheses must appear in this form.

Execution of this statement causes records to be written on the device referenced by u. The contents of the records are the values taken sequentially from the list, k, converted according to the format specification, f.

The form of an unformatted *WRITE* statement is:

WRITE (u) k.

The verb *WRITE* and the parentheses must appear in this form.

Execution of this statement causes binary information from the list, k, to be written in records on the unit defined by u.

Examples

```
WRITE (1, 4) A, B, C
WRITE (7) R, S, T
WRITE (K, 12) X, (Y (J), J = 1, M), I
WRITE (N) W, Z, (F (K), K = 1, 5)
```

Several records may be written with a single *WRITE* statement. The number of records is determined by the list and format specifications. Successive records are written until the data are exhausted. If the data do not fill a record, the record is filled with blanks.

6.7 REWIND STATEMENT

This statement is of the form:

REWIND u.

Execution of this statement causes the physical unit defined by u to be rewound.



SECTION 6 INPUT/OUTPUT STATEMENTS

6.8 BACKSPACE STATEMENT

This statement has the form:

BACKSPACE u.

The *BACKSPACE* statement causes the physical unit defined by u to be backspaced one record.

6.9 ENDFILE STATEMENT

This statement has the form:

ENDFILE u.

When this statement is executed, a file mark is written on the physical unit defined by u.

6.10 FORMAT STATEMENTS

FORMAT statements, with input/output operations, specify conversion and editing of information between program storage and external representation. *FORMAT* statements are nonexecutable and must have a statement label to be referenced by input/output statements. Conversion performed according to a *FORMAT* statement during output is in general the reverse of conversion performed during an input operation.

A *FORMAT* statement is expressed as:

n FORMAT (f1, f2, f3, ..., fn)

where

n is the statement label and the fn are field specifications

The noun *FORMAT* and the parentheses must appear in this form.

When formatted records are output to a printer, the first character of the record is not printed but is processed as a printer vertical spacing control character as follows:

Character	Vertical Spacing
blank	One line
0	Two lines
1	To first line of next page



SECTION 6 INPUT/OUTPUT STATEMENTS

6.11 FIELD SPECIFICATIONS

FIELD specifications describe the type of conversion and editing to be performed on each variable appearing in the input/output list. *FIELD* specifications can be in any of the following forms:

rAw rFw.d rEw.d rDw.d rlw nHs nX rLw rGw.d

where:

- a. The characters A, D, E, F, G, L, and I indicate the manner of conversion for variables in the list.
- b. The characters H and X represent characters to be input/ output directly from the format.
- c. The character / represents the end of a record.
- d. w and n are nonzero integer constants defining the width of the field (including digits, decimal points, and algebraic signs) in the external character string.
- e. d is an integer specifying the number of fractional digits appearing in the external string.
- f. r is an optional, nonzero integer indicating that the specification is to be repeated r times.
- g. s is a string of acceptable **FORTRAN** characters.

6.12 F CONVERSION

Form

rFw.d

Only real data may be processed by this form of conversion.



SECTION 6 INPUT/OUTPUT STATEMENTS

6.12.1 Output

The field is right-justified with as many leading blanks as necessary to fill w. Negative values are preceded by a minus sign. Internal values are converted to fixed-point decimal numbers and rounded to d decimal places.

For a field specification of F10.4:

368.4	is converted to	368.4000
12.0	is converted to	12.0000
-17.90767	is converted to	-17.9077
37.5E-2	is converted to	.3750

If a value requires more positions than allowed by w, the most significant digits, including sign if negative, are output. The error indication is designated by an asterisk in the least significant character position.

For a field specification of F6.4:

4739.76	is converted to	4740*
-12.463	is converted to	-12.5*

6.12.2 Input

Input strings are decimal numbers of length w with d characters in the fractional portion. Blanks are treated as zeros. If a decimal point is present in a value, the fractional portion of the value is explicitly defined by that decimal point character.

For a field specification F8.3:

35	is converted to	0.035
964372	is converted to	964.372
0.53821	is converted to	0.53821
-16.402	is converted to	-16.402
-12	is converted to	-0.012
47.E-4	is converted to	0.0047



SECTION 6

INPUT/OUTPUT STATEMENTS

6.13 E CONVERSION

Form

rEw.d

Only real data may be processed by this form of conversion.

6.13.1 Output

Internal values are converted to decimal values of the forms:

.ddd...dE + ee and .ddd...E-ee

where:

ddd...d represents d digits, and
ee is a decimal exponent.

The leading decimal point and E characters are present exactly as shown. Internal values are rounded to d digits, and negative values are preceded by a minus sign. The external field is right-justified and preceded by blanks to fill the width, w. This field width includes the exponent digits, the sign of the exponent (minus or space), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or space). This means that the field width should correspond to the relation: $w \geq d + 6$.

If w is less than (d + 6), the format is in error.

For a field specification of E12.5:

76.573	is converted to	.76573E 02
58796.341	is converted to	.58796E 05
-369.7583	is converted to	-.36976E 03
0.006873	is converted to	.68730E-02
0.2	is converted to	.20000E 00
-0.0000054	is converted to	-.54000E-05

6.13.2 Input

Each external value is of field width w with d characters in the fractional part of the value. The value is right-justified with all blanks counting as zeros. A minus sign may precede the value of the exponent. A decimal point placed in the fractional part takes precedence over the d specification. The character E may be present to separate the value and the exponent.



SECTION 6 INPUT/OUTPUT STATEMENTS

For a field specification of E10.3:

123E3	is converted to	123.0
12874E2	is converted to	1287.4
-563E-02	is converted to	-0.00563
398E00	is converted to	0.398
5387601	is converted to	538.76
5455-01	is converted to	0.5455
-6.7563E05	is converted to	-675630.0

6.14 D CONVERSION

The *D* conversion is used for the input/output of double-precision numbers. It is used exactly as the *E* conversion except the letter *E* is replaced by *D*.

6.15 I CONVERSION

Form

rlw

Only integer data may be processed by this form of conversion.

6.15.1 Output

Internal values are converted to integer constants. Negative values are preceded by a minus sign. Each field is right-justified and filled with leading blanks.

For a field specification of I6:

281	is converted to	281
-3567	is converted to	-3567

If the data require more character positions than allowed by the width, *w*, only the most significant *w* positions are output.

For a field specification of I3:

281	is converted to	3*
-6374	is converted to	-6*



SECTION 6

INPUT/OUTPUT STATEMENTS

6.15.2 Input

External input values are right-justified with the width, *w*. Blanks are counted as zeros. Input values must be integer values. A preceding minus sign may be placed on a value.

For a field specification of I4:

120	is converted to	120
-144	is converted to	-144
1 2	is converted to	102
3	is converted to	-3

6.16 A CONVERSION

An A format conversion is used in conjunction with a *READ* or *WRITE* statement for the input/output of alphanumeric information to or from a *REAL* list element. The general form is *rAw*, where *r* and *w* are unsigned integer constants. If *r* is one, it can be omitted.

On input, *rAw* will be interpreted to mean that the next *r* successive fields of *w* characters are each to be stored in the associated *REAL* list elements. If *w* is greater than 4, only the four right-most characters will be significant. If *w* is 4 or less, the characters will be left-justified, and the word(s) filled with blanks, if necessary.

On output, *rAw* will be interpreted to mean that the next *r* successive fields of *w* characters are each to be the result of alphanumeric transmission from the specified *REAL* list elements. If *w* exceeds 4, only four characters of output will be transmitted, preceded by *w* - 4 blanks. If *w* is 4 or less, the *w* left-most characters of the specified storage element will be transmitted.

6.17 H CONVERSION

In **FORTRAN**, Hollerith information consists of the legal **FORTRAN** character set plus the additional characters

" , # , : , ; , ! , % , & , ' ,

Information input from the typewriter or paper tape is converted to an internal code used by **FORTRAN**. When this information is output, the internal codes are converted to the appropriate typewriter or paper tape codes.

Form: nHs.



SECTION 6

INPUT/OUTPUT STATEMENTS

6.17.1 Output

The number of characters, n , in the string, s , should contain exactly the number of characters specified so that characters from other fields are not taken as part of the string.

Blanks are counted as characters in the string.

Examples

Specification

1HR
8HbSTRINGb
11HX(1,3) = 12.0

External Output

R
bSTRINGb
X(1,3) = 12.0

6.17.2 Input

The w characters in the string, s , are replaced by the next w characters from the input record. The result is a new string in the field specification.

Example

Specification

5H12345
7HbTRUEbb
8Hbbbbbbbbb

Input String

ABCDE
FALSEbb
MATRIXbb

Resultant Specification

5HABCDE
7HFALSEbb
8HMATRIXbb

This feature can be used to change titles, dates, headings, etc., that are output with the program data.



SECTION 6

INPUT/OUTPUT STATEMENTS

6.18 X SPECIFICATION

Form: nX.

This specification causes no conversion. On output, n blanks are inserted in the external record. On input, n spaces are skipped from the input record.

Output Example

Specification	Output
1HA, 4X, 2HBC	AbbbbBC
4X, 3HABC	bbbbABC
1X, 3HABC, 3X	bABCbbb

Input Example

Specification	Input String	Resultant Input
F4.1, 3X, F3.0	12.5RRR120	12.5,120.

The RRR characters are ignored by the 3X specification.

6.19 L FORMAT CODE

General Form

rLw

where:

r is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

w is an unsigned integer constant that specifies the number of characters of data.

Logical variables may be read or written by means of the format code Lw.



SECTION 6 INPUT/OUTPUT STATEMENTS

On input, the first T or F encountered in the next *w* characters of the input record causes a value of .TRUE. or .FALSE., respectively, to be assigned to the corresponding logical variable. If field *w* consists entirely of blanks, a value of .FALSE. is assumed.

On output, a T or F is inserted in the output record corresponding to the value of the logical variable in the I/O list. The single character is preceded by *w* - 1 blanks.

6.20 G FORMAT CODE

General Form

rGw.d

where:

r is optional and is an unsigned integer constant used to denote the number of times the same format code is repetitively referenced.

w is an unsigned integer constant specifying the total field length.

d is an unsigned integer constant specifying the number of significant digits.

The G format code is a generalized code in that it automatically selects an output format appropriate to the magnitude of the real data.

Input processing is the same as for the *F* conversion.

The *w* portion of the G format code reserves the four right-most positions for a decimal exponent field.

If the real data, *n*, are in the range $0.1 \leq n < 10^d$, where *d* is the *d* portion of the format code *Gw.d*, then this exponent field is blank. Otherwise, the real data are transferred with an E or D decimal exponent depending on the type of the real data.

For the purpose of simplification, the following examples deal with the printed line. However, the concepts apply to all input/output media.

**SECTION 6**
INPUT/OUTPUT STATEMENTS**Example 1**

Assume that the variables A, B, C, and D are of type real whose values are 292.7041, 82.43441, 136.7632, 0.8081945, respectively.

```
1  FORMAT  (G12.4,G12.5,G12.4,G12.7)
2  FORMAT  (G13.4,G13.5,G13.4)
3  FORMAT  (G13.4)
.
.
.
WRITE  (0, n) A, B, C, D
.
.
.
```

Explanation:

- a. If n has been specified as 1, the printed output would be as follows (b represents a blank):

Print Position 1	Print Position 48
↑	↑
bbb292.7bbbbbb82.434bbbbbbb136.8bbbb.8081945bbbb	

- b. If n has been specified as 2, the printed output would be:

Print Position 1	Print Position 39	
↑	↑	
bbbb292.7bbbbbb82.434bbbbbbb136.8bbbb		Line 1
bbbb.8082bbbb		Line 2

From the above example, it can be seen that by increasing the field width reserved (w), blanks are inserted.

- c. If n has been specified as 3, the printed output would be:

Print Position 1	
↑	
bbbb292.7bbbb	Line 1

(continued)



SECTION 6 INPUT/OUTPUT STATEMENTS

bbbb82.43bbbb	Line 2
bbbb136.8bbbb	Line 3
bbbb.8082bbbb	Line 4

From the above example, it can be seen that the same format code is used for each variable in the list. Each repetition of the same format code causes a new line to be printed.

6.21 SCALE FACTOR P

The representation of the data, internally or externally, can be modified by the use of a scale factor followed by the letter P preceding the F, E, G, and D format codes.

The scale factor affects the appropriate conversions in the following manner:

- For F, E, G, and D input conversions (provided no exponent exists in the external field) and F output conversions, the scale factor effect is as follows:

externally represented number equals internally represented number times the quantity ten raised to the nth power
- For F, E, G, and D input, the scale factor has no effect if there is an exponent in the external field.
- For E and D output, the basic real constant part of the quantity is multiplied by ten to the nth power and the exponent is reduced by the scale factor.
- For G output, the effect of the scale factor is suspended unless the magnitude of the datum to be converted is outside the range that permits the effective use of F conversion. If the effective use of E conversion is required, the scale factor has the same effect as with E output.

For example, if input data are in the form xx.xxxx and it is desired to use this internally in the form .xxxxxx, the format code used to effect this change is 2PF7.4.



SECTION 6

INPUT/OUTPUT STATEMENTS

6.21.1 Input

As another example, consider the following input data:

27bbb-93.2094bb-175.8041bbbb55.3647

where b represents a blank.

The following statements:

```
5      FORMAT      (I2,3F11.4)
      .
      .
      .
      READ          (0,5) K,A,B,C
```

cause the variables in the list to assume the following values:

K : 27	B : -175.8041
A : -93.2094	C : 55.3647

The following statements:

```
5      FORMAT      (I2,1P3F11.4)
      .
      .
      .
      READ          (0,5) K,A,B,C
```

cause the variables in the list to assume the following values:

K : 27	B : -17.58041
A : -9.32094	C : 5.53647

The following statements:

```
5      FORMAT      (I2,-1P3F11.4)
      .
      .
      .
      READ          (0,5) K,A,B,C
```



SECTION 6 INPUT/OUTPUT STATEMENTS

causes the variables in the list to assume the following values:

K : 27	B : -1758.041
A : -932.094	C : 553.647

6.21.2 Output

Assume the variables K,A,B, and C have the following values:

K : 27	B : -175.8041
A : -93.2094	C : 55.3647

then the following statements:

```
5      FORMAT      (I2,1P3F11.4)
      .
      .
      .
      WRITE      (0,5) K,A,B,C
```

cause the variables in the list to output the following values:

K : 27	B : -1758.041
A : -932.094	C : 553.647

The following statements:

```
5      FORMAT      (I2,-1P3F11.4)
      .
      .
      .
      WRITE      (0,5) K,A,B,C
```

cause the variables in the list to output the following values:

K : 27	B : -17.5804
A : -9.3209	C : 5.5365

For output, when scale factors are used, they have effect only on real data. However, this real data may contain an E or D decimal exponent. A positive scale factor used with real



SECTION 6

INPUT/OUTPUT STATEMENTS

data that contains an E or D decimal exponent increases the number and decreases the exponent. Thus, if the real data were in a form using an E decimal exponent and the statement `FORMAT (1X,I2,3E13.3)` used with an appropriate `WRITE` statement resulted in the following printed line:

b27bbbb-932Eb02bbbb-175Eb03bbbb.553Eb02

the statement `FORMAT (1X,I2,1P3E13.3)` used with the same `WRITE` statement results in the following printed output:

b27bbb-9.321Eb01bbb-1.758Eb02bbbb5.536Eb01

The statement `FORMAT (1X,I2,1P3E13.3)` used with the same `WRITE` statement results in the following printed output:

27bbbb-.093Eb03bbbb-.018Eb04bbbb.055Eb03

The scale factor is assumed to be zero if no other value has been given. However, once a value has been given, it will hold for all format codes following the scale factor within the same `FORMAT` statement. This also applies to format codes enclosed within an additional pair of parentheses.

6.22 / SPECIFICATION

Form

/.

Each slash (/) specified in the format causes the termination of a record and processing of the next record. Successive slashes (///...//) cause successive records to be ignored on input, and successive blank records to be written on output. A slash separating two field specifications removes the need for a comma separator. For example,
F5.4/4F10.3.

Output Example

For a specification (1HA/1HB/1HC/1HD) the resultant output records are:

A
B
C
D



SECTION 6 INPUT/OUTPUT STATEMENTS

Input Example

Using the four records output from the previous example, an input specification (1H1/1H2//1H3) produces the resultant specification (1HA/1HB//1HD).

6.23 REPEAT SPECIFICATIONS

The A, D, F, E, and I field specifications can be repeated by using the repeat count *r* in the forms *rAw*, *rDw*, *rFw.d*, *rEw.d*, *rlw*, *rLw*, and *rGw.d*.

Examples

4F10.5,F3.6 is equivalent to F10.5,F10.5,F10.5,F10.5,F3.6

2F4.1,2E7.1 is equivalent to F4.1,F4.1,E7.1,E7.1

2F5.2,3I6,2E8.2 is equivalent to F5.2,F5.2,I6,I6,I6,E8.2,E8.2

Repetition of a group of field specifications is accomplished by enclosing the group in parentheses preceded by an integer repeat count. If no repeat count is specified, the count is taken as one.

Examples

2(F10.5,I6) is equivalent to F10.5,I6,F10.5,I6

2(E9.3,F7.1/I4) is equivalent to E9.3,F7.1/I4,E9.3,F7.1/I4

3(4F5.0,2E8.2) is equivalent to 4F5.0,2E8.2,4F5.0,2E8.2,
4F5.0,2E8.2

Example

50 FORMAT (4X,2(I5,6F8.2)//)



SECTION 6

INPUT/OUTPUT STATEMENTS

The use of additional parentheses (up to two levels) within a *FORMAT* statement is permitted to enable the user to repeat the same format code when transmitting data. For example, the statement:

```
10 FORMAT (2(G10.6,G7.1),G4)
```

is equivalent to

```
10 FORMAT (G10.6,G7.1,G10.6,G7.1,G4)
```

If the data exists with a D decimal exponent, it is transferred with this D decimal exponent.

If a multiline listing is desired such that the first two lines are to be printed according to a special format and all remaining lines according to another format, the last format code in the statement should be enclosed in a second pair of parentheses. For example, in the statement:

```
FORMAT(G2,2G3.1/G10.8/(3G5.1))
```

If more data items are to be transmitted after the format codes have been completely used, the format repeats from the last left parenthesis. Thus, the printed output would take the following form:

```
G2,G3.1,G3.1
      G10.8
G5.1,G5.1,G5.1
G5.1,G5.1,G5.1
      .
      .
      .
```

As another example, consider the following statement:

```
FORMAT(G2/2(G3,G6.1),G9.7)
```

If 13 data items are to be transmitted, the printed output on a *WRITE* statement takes the following form:



SECTION 6 INPUT/OUTPUT STATEMENTS

G2
G3,G6.1,G3,G6.1,G9.7
G3,G6.1,G3,G6.1,G9.7
G3,G6.1

6.24 FORMAT CONTROL AND LINE INTERACTION

Execution of a formatted *READ* or *WRITE* statement initiates format control. The conversion performed on data depends on information jointly provided by the next element of the input/output list and the next field specification of the *FORMAT* statement. If there is a list, at least one field specification of type D, E, F, G, L, A, or I should be present in the *FORMAT* statement.

Execution of a formatted *READ* statement causes one record to be input. Each D, E, F, G, L, A, or I specification has a corresponding element in the list. Each H or X specification has no corresponding element in the list and the format control communicates information directly to the record. When a slash is encountered or the entire input record is processed, the record is terminated. If more input is necessary, the next record is input. Any unprocessed characters of a record are skipped when a slash is encountered.

A *READ* statement is terminated upon ending the list if:

- a. The next specification is A, D, E, F, G, I, or L.
- b. The format control has reached the last outer right parenthesis of the *FORMAT* statement.

If the list ends and the next specification is an H or X, data are processed (with the possibility of additional records being input) until one of the two above conditions is met.

If the format control reaches the right-most parenthesis of the *FORMAT* statement and more list remains to be processed, the following steps are taken:

- a. A new record is input and remaining data in the previous record ignored.
- b. Format control reverts to the point immediately following the last left parenthesis.

If group repeat specifications exist in the format, this point is at the right-most group of the format. The repeat count is not taken into consideration. If no groups are present, the format is started from the beginning.



SECTION 6

INPUT/OUTPUT STATEMENTS

When a formatted *WRITE* statement is executed, records are written each time 120 characters have been processed, a slash is encountered, or the format control terminates. The format control terminates by one of the two methods described for *READ* termination. Incomplete records are filled with blanks to maintain standard record lengths.



SECTION 7 PROGRAMS AND SUBPROGRAMS

7.1 GENERAL

An executable **FORTRAN** program consists of a main program and any required subprograms. Subprograms may be defined by the programmer or contained in the system library. Each program or subprogram must contain at least one executable statement.

7.2 MAIN PROGRAMS

A main program is a program unit consisting of a set of **FORTRAN** statements, comment lines, and an *END* line. The program may be preceded by specification statements.

A main program cannot contain a subprogram definition statement, namely:

- a FUNCTION statement
- a SUBROUTINE statement
- a BLOCK DATA statement

A main program may contain calls to other subprograms or may contain statement function subprograms.

An MOS main program can accept a main program entry name definition of the following format:

NAME N1, N2, ..., Nn

where:

N1, N2, ..., Nn are entry names by which the main program
can be referenced

7.3 SUBPROGRAMS

Subprograms are program units which may be called by other programs or subprograms. Subprograms are categorized as one of the following:

- PROCEDURE SUBPROGRAMS
- FUNCTION subprogram
- SUBROUTINE subprogram



SECTION 7 PROGRAMS AND SUBPROGRAMS

SPECIFICATION subprogram
BLOCK DATA subprogram

Functions are programmed procedures that are often used to provide solutions to mathematical functions. Function references may be used in the same manner as references to variables in an expression. For example: $X = AB * \text{SIN}(Y) + C * \text{COS}(Y * Z)$, where **SIN** is the name of the sine function, **COS** is the name of the cosine function, and (Y) and $(Y * Z)$ are their respective argument lists. The value returned for a function reference is of the same mode as the function name, corresponding to the rules for real and integer symbolic names.

7.3.1 Function Subprograms

A function subprogram is defined external to the program unit by which it is referenced. A function subprogram is defined by having as its first statement, other than comment lines, a statement of the form:

```
FUNCTION f(a1, a2, a3, ..., an)
```

where

f is the symbolic name of the function and

a_i represent dummy arguments.

Each a_i is either a variable name or an array name. The a_i defines the type, number, and order of the *FUNCTION* arguments. A function subprogram must have at least one argument.

A function subprogram is executed at the first executable statement following the *FUNCTION* statement. Specification statements (*DIMENSION*, *COMMON*, and *EQUIVALENCE*) may immediately follow the *FUNCTION* statement. If present, these must precede any other statement, excluding comments. The symbolic names of the dummy arguments, a_i , may not appear in an *EQUIVALENCE* or *COMMON* statement.

A function subprogram must contain at least one *RETURN* statement, and the last statement executed in a *FUNCTION* must be a *RETURN* statement. The function subprogram is ended by an *END* line.



SECTION 7 PROGRAMS AND SUBPROGRAMS

The symbolic name, *f*, of the *FUNCTION* must appear as a variable name within the subprogram. The value returned for a *FUNCTION* is the last value assigned to this name prior to execution of a *RETURN* statement. The type of the *FUNCTION* value is as for a variable (section 2.3.1).

The symbolic name of the function must not appear in any nonexecutable statement within the subprogram.

Example

```
FUNCTION XP(A,B,I)
  DIMENSION B(10)
  XP = 0.
  DO 1 J = 1,10
1    XP = (A*B(J))**I + XP
  RETURN
END
```

A *FUNCTION* is executed with a function reference by a main program or another subprogram. The actual arguments in the call must correspond in type, number, and order with the *FUNCTION* dummy arguments. If a dummy argument of a *FUNCTION* is an array name, the corresponding actual argument must be an array name.

Example:

A call for the example *FUNCTION* shown above would be:
 $W = XP(R,S,K)$, where *S* is an array.

Type Specification of *FUNCTION* Subprogram

In addition to declaring the type of a *FUNCTION* name by the predefined convention, there exists the option of explicitly specifying the type of a *FUNCTION* name within the function statement.

General Form

Type *FUNCTION* name (*a*₁, *a*₂, *a*₃, ..., *a*_{*n*})

where:

Type is integer, real, double precision, complex, or logical.



SECTION 7 PROGRAMS AND SUBPROGRAMS

name is the name of the *FUNCTION* subprogram.

a1, a2, a3,...,an are nonsubscripted variables, or dummy names of subroutine or other *FUNCTION* subprograms. (There must be at least one argument in the argument list.)

Example 1

```
REAL FUNCTION SOMEF (A,B)
.
.
.
SOMEF = A**2 + B**2
.
.
.
RETURN
END
```

Example 2

```
INTEGER FUNCTION CALC (X,Y,Z)
.
.
.
CALC = X + Y + Z**2
.
.
.
RETURN
END
```

Explanation:

The *FUNCTION* subprograms SOMEF and CALC in Examples 1 and 2 are declared as type *REAL* and *INTEGER*, respectively.



SECTION 7 PROGRAMS AND SUBPROGRAMS

7.3.2 Subroutine Subprograms

A subroutine subprogram is defined external to the program unit that references it. Subroutines, unlike functions, do not have values associated with them and cannot be referenced in an expression. Subroutines are accessed by *CALL* statements.

A subroutine subprogram is defined by having as its first statement, other than comment lines, a statement of the form: *SUBROUTINE S (a1, a2, a3, ..., an)* or *SUBROUTINE S*, where *S* is the symbolic name of the subroutine and *a_i* represents dummy arguments of the subroutine. Each *a_i* is either a variable or an array name, or the name of an external procedure. If no arguments are passed to the subroutine, the second form is used.

The symbolic name of the subroutine must not appear in any statement in the subprogram. The symbolic names of the dummy arguments may not appear in *COMMON* or *EQUIVALENCE* statements.

A subroutine is executed at the first executable statement. Specification statements must immediately follow the *SUBROUTINE* statement and precede any executable statement. A subroutine must have at least one *RETURN* statement. The last statement executed by a subroutine must be a *RETURN* statement.

Example

```
SUBROUTINE R(A,I,Z)
  DIMENSION A(10)
  Z = 0
  DO 1 J = 1,10
1    Z = Z + A(J)**I
  RETURN
  END
```

A subroutine is referenced with a *CALL* statement. The argument list in the reference must agree in type, number, and order with the dummy arguments of the subroutine. If a dummy argument is an array name, the corresponding actual argument must be an array name.

Example:

A call for the example *SUBROUTINE* above would be: *CALL R (T,K,D)* where *T* is an array.



SECTION 7 PROGRAMS AND SUBPROGRAMS

7.3.3 Block Data Subprogram

To initialize variables in a *COMMON* block, a separate subprogram must be written. This separate subprogram contains only the *DATA*, *COMMON*, *DIMENSION*, *EQUIVALENCE*, and *TYPE* statements associated with the data being defined. Data may be initialized in labeled (named), but not unlabeled, *COMMON* by the *BLOCK DATA* subprogram.

General Form

```
BLOCK DATA
.
.
.
END
```

- The *BLOCK DATA* subprogram may not contain any executable statements.
- The *BLOCK DATA* statement must be the first statement in the subprogram.
- All elements of a *COMMON* block must be listed in the *COMMON* statement, even though they are not all initialized; for example, the variable *A* in the *COMMON* statement in the following example does not appear in the data initialization statement.

```
BLOCK DATA
COMPLEX C
COMMON/ELN/C,A,B/RMG/Z,Y
DATA C/(2.4.3.769)/
```

- Data may be entered into more than one *COMMON* block in a single *BLOCK DATA* subprogram.

7.3.4 Data Initialization Statement

General Form

```
DATA k1,...,kn/j1*d1,...,jn*dn/,kn + 1,...,k/jn + 1*dn + 1,...,j*d/,...
```



SECTION 7 PROGRAMS AND SUBPROGRAMS

where:

$k_1, \dots, k$ are variables and/or subscripted variables (in which case the subscripts must be integer constants), or array elements, array names, or implied DO lists

$d_1, \dots, d$ are values representing integer, real, double-precision, complex, logical or Hollerith data constants.

$j_1^*, \dots, j^*$ represent unsigned integer constants indicating the number of consecutive variables that are to be assigned the value of $d_1, \dots, d_n$.

A data initialization statement defines initial values of variables and array elements. There must be a one-to-one correspondence between these variables (i.e., $k_1, \dots, k$) and the data constants (i.e., $d_1, \dots, d$).

Example 1

```
DIMENSION D(10)
```

```
DATA A,B,C/5.0,6.1,7.3/,D(1),D(2),D(3),D(4),D(5)/5*1.0/
```

Explanation:

The *DATA* statement indicates that the variables A, B, and C are to be initialized to the values 5.0, 6.1, and 7.3, respectively. In addition, the statement specifies that the first five variables in array D are to be initialized to 1.0.

Example 2

```
DIMENSION A(5),B(3),L(2)
```

```
DATA A(1),A(2),A(3),A(4),A(5)/5*1.0/,B(1),B(2)/2*5.0/,L(1),L(2)/.TRUE.,.FALSE./
```

Explanation:

The *DATA* statement specifies that all the variables in array A are to be initialized to 1.0 and the first two elements of array B are to be initialized to 5.0. All the logical variables, (L(1) and L(2)), in array L are initialized to .TRUE. and .FALSE., respectively.



SECTION 7 PROGRAMS AND SUBPROGRAMS

An initially defined variable, or any element, may not be in blank common. However, in a labeled common block, they may be initially defined only in a block data subprogram. (See the section Subprograms.)

Example 3

```
DIMENSION A(3), B(3,2)
DATA A/1.0,2.0,3.0/,((B(I,J),J = 1,2),I = 1,3)/6*5./
```

Explanation:

The *DATA* statement loads real numbers 1.0, 2.0, and 3.0 into array A. It also loads real number 5. into every element of array B.

7.4 STATEMENT FUNCTIONS

A statement function is defined internal to the program unit in which it is referenced. All statement functions must precede the first executable statement and must follow any specification statements of the program unit.

A statement function is defined in a single expression of the form: $f(a_1, a_2, a_3, \dots, a_n) = e$, where f is the function name, a_i represents arguments, and e is an expression. The resultant value of the function is either a real or integer value corresponding to the function name. The a_i are distinct variable names and are called dummy arguments. They serve to indicate the type, number, and order of the function arguments. The expression e is an arithmetic expression and may contain references to previously defined statement functions.

A statement function is referenced by a function call, $f(a_1, a_2, a_3, \dots, a_n)$, appearing in an arithmetic expression. A statement function may be referenced only within the program unit in which it is defined. The arguments used in the reference must agree in type, number, and order with the corresponding dummy arguments.

Example

The statement function:

$$SF(X) = A * X ** 2 + B * X + C$$

can be referenced in the program by:

$$W = SF(Y)$$



SECTION 7 PROGRAMS AND SUBPROGRAMS

7.5 INTRINSIC FUNCTIONS

Intrinsic functions are commonly used subprograms contained in the **FORTRAN** library. The symbolic names and meanings of the intrinsic functions are shown in table 7-1.

An intrinsic function is referenced by a function call in an arithmetic expression. The arguments in the argument list must agree in type, number, and order with those shown in table 7-1.

Example

```
      IF (SIGN(W,X)) 1,2,2
1     W = ABS(X)- ABS(Y)
2     S = W*FLOAT(I*J)
      K = IFIX(X)+J
```

7.6 BASIC EXTERNAL FUNCTIONS

Basic external *FUNCTIONS* are standard subprograms contained in the **FORTRAN** library. These are referenced in the same manner as normal *FUNCTIONS*. The symbolic names and meanings of the basic external *FUNCTIONS* are shown in table 7-2.

7.7 DUMMY ARGUMENTS

Dummy arguments provide a means of passing information between a subprogram and the program or subprogram that called it. Both function and subroutine subprograms may have dummy arguments. A subroutine need not have any, while a function must have at least one. Dummies provide definitions of the data type, number, and sequence of subprogram parameters.

A dummy can be classified within a subprogram as a variable, an array, or an external procedure name. The actual arguments defined by a calling program or subprogram to which a dummy can correspond are: Hollerith constants, variables, array elements, arrays, expressions, and external procedure names.

Within a subprogram, a dummy can be used in much the same way as any other variable or array. A dummy can not appear in a *COMMON* or *EQUIVALENCE* statement.



SECTION 7

PROGRAMS AND SUBPROGRAMS

The actual arguments (except for Hollerith constants) used in a calling statement agree in data type with the corresponding dummy arguments, that is, real to real, integer to integer, and array to array. If an actual argument is an expression, the result of the expression should correspond in data type to the dummy.

A dummy array is defined as an argument which appears in a *DIMENSION* statement in the subprogram. A dummy array does not occupy any storage but tells the subprogram that the argument supplied in the calling statement defines the first element of an actual array. The calling argument need not have the same dimensions as the dummy array. Useful operations can sometimes be performed by defining different dimensions for the dummy and calling arguments.

Example

DIMENSION	A(10,10)
CALL	FM(A(6,1))
.	
.	
.	
SUBROUTINE	FM(B)
DIMENSION	B(50)

For this case, one-dimensional dummy array B corresponds to the last half of two-dimensional array A. If the calling statement were CALL FM (A), dummy array B would correspond to the first half of array A.

7.8 ADJUSTABLE DIMENSIONS

As shown in the previous examples, the maximum value of each subscript in an array is specified by a numeric value. These numeric values (maximum value of each subscript) are known as the absolute dimensions of an array and may never be changed. However, if an array is used in a subprogram (section 7.3) and is not in *COMMON*, the size of this array does not have to be explicitly declared in the subprogram by a numeric value. That is, the specification statement, appearing in a subprogram, may contain integer variables that specify the size of the array. These integer variables must be either actual or implicit subprogram arguments. When the subprogram is called, these integer variables receive their values from the calling program. Thus, the dimensions (size) of a dummy array appearing in a subprogram are adjustable and may change each time the subprogram is



SECTION 7 PROGRAMS AND SUBPROGRAMS

called. Integer variables that provide dimension information may not be redefined within the subprogram.

The absolute dimensions of an array must be declared in a calling program. The adjustable dimensions of an array, appearing in a subprogram, should be less than or equal to the absolute dimensions of that array as declared in the calling program.

The following example illustrates the use of adjustable dimensions.

Example

CALLING PROGRAM

```
DIMENSION A(5,5)
.
.
.
CALL MAPMY(...,A,2,3,...)
.
.
.
```

SUBPROGRAM

```
SUBROUTINE MAPMY
(...,R,L,M,...)
.
.
.
.
DIMENSION...,R(L,M),...
.
.
.
DO 100 I = 1,L
.
.
.
```

Explanation:

The statement DIMENSION A(5,5) appearing in the calling program declares the absolute dimensions of array A. When subroutine MAPMY is called, dummy argument R assumes array name A and dummy arguments L and M assume the values 2 and 3, respectively. The correspondence between the subscripted variables of arrays A and R is shown in the following example.

R(1,1)R(2,1)R(1,2)R(2,2)R(1,3)R(2,3)

A(1,1)A(2,1)A(3,1)A(4,1)A(5,1)A(1,2)A(2,2)...



SECTION 7

PROGRAMS AND SUBPROGRAMS

Thus, in the calling program the subscripted variable $A(1,2)$ refers to the sixth subscripted variable in array A. However, in subprogram MAPMY, the subscripted variable $R(1,2)$ refers to the third subscripted variable in array A, namely, $A(3,1)$. This is so because the dimensions of array R as declared in the subprogram are not the same as those in the calling program.

If the absolute dimensions in the calling program were the same as the adjusted dimensions in the subprogram, the subscripted variables $R(1,1)$ through $R(5,5)$ in the subprogram would always refer to the same storage locations as specified by the subscripted variables $A(1,1)$ through $A(5,5)$ in the calling program, respectively.

The numbers 2 and 3, which became the adjusted dimension of dummy array R, could also have been variables in the argument list or implicit arguments in a *COMMON* block. For example, assume that the following statement appeared in the calling program.

```
CALL MAPMY (...,A,I,J,...)
```

Then as long as the values of I and J are previously defined, the arguments may be variables. In addition, the variable dimension size may be passed through more than one subprogram level. For example, the subprogram MAPMY could have contained a call statement to another subprogram in which dimension information about A could have been passed.

Dummy variables (e.g., L and M) may be used as dimensions of an array only in a *FUNCTION* or *SUBROUTINE* subprogram.

7.9 EXTERNAL STATEMENT

General Form

```
EXTERNAL a,b,c,...
```

where:

a,b,c,... are names of subprograms that are passed as arguments to other subprograms.

The name of any subprogram passed as an argument to another subprogram must appear in an *EXTERNAL* statement in the calling program. For example, assume that SUB and MULT are subprogram names in the following statements:



SECTION 7 PROGRAMS AND SUBPROGRAMS

Example

CALLING PROGRAM

```
EXTERNAL MULT  
.  
.  
.  
CALL SUB(J,MULT,C)  
.  
.  
.
```

SUBPROGRAM

```
SUBROUTINE SUB(K,Y,Z)  
IF(K)4,6,6  
4 D = Y(K,Z**2)  
.  
.  
.  
6 RETURN  
END
```

Explanation:

In this example, the subprogram name MULT is used as an argument in subprogram SUB. The subprogram name MULT is passed to dummy variable Y, and variables J and C are passed to dummy variables K and Z, respectively. Subprogram MULT is called and executed only if the value of K is negative.



SECTION 7

PROGRAMS AND SUBPROGRAMS

Table 7-1. Basic Intrinsic Functions

Intrinsic Function	Definition	Arguments	Name	Type of Argument	Type of Function
Absolute Value	$ a $	1	ABS IABS DABS	Real Integer Double	Real Integer Double
Truncation	Sign of a times largest integer $\leq a $	1	AIN INT IDINT	Real Real Double	Real Integer Integer
Remaindering*	$a_1 \pmod{a_2}$	2	AMOD MOD	Real Integer	Real Integer
Choosing Largest Value	$\text{Max}(a_1, a_2, \dots)$	≥ 2	AMAX0 AMAX1 MAX0 MAX1 DMAX1	Integer Real Integer Real Double	Real Real Integer Integer Double
Choosing Smallest Value	$\text{Min}(a_1, a_2, \dots)$	≥ 2	AMIN0 AMIN1 MIN0 MIN1 DMIN1	Integer Real Integer Real Double	Real Real Integer Integer Double
Float	Conversion from integer to real	1	FLOAT	Integer	Real



SECTION 7
PROGRAMS AND SUBPROGRAMS

Table 7-1. Basic Intrinsic Functions (continued)

Intrinsic Function	Definition	Arguments	Name	Type of Argument	Type of Function
Fix	Conversion from real to integer	1	IFIX	Real	Integer
Transfer of Sign	Sign of a_2 times $ a_1 $	2	SIGN ISIGN DSIGN	Real Integer Double	Real Integer Double
Positive Difference	$a_1 - \min(a_1, a_2)$	2	DIM IDIM	Real Integer	Real Integer
Obtain Most Significant Part of Double Precision Argument		1	SNGL	Double	Real
Obtain Real Part of Complex Argument		1	REAL	Complex	Real
Obtain Imaginary Part of Complex Argument		1	AIMAG	Complex	Real
Express Single Precision Argument in Double Precision Form		1	DBLE	Real	Double
Express Two Real Arguments in Complex Form	$a_1 + a_2 \sqrt{-1}$	2	CMPLX	Real	Complex



SECTION 7

PROGRAMS AND SUBPROGRAMS

Table 7-1. Basic Intrinsic Functions (continued)

Intrinsic Function	Definition	Arguments	Name	Type of Argument	Type of Function
Obtain Conjugate of a Complex Argument			CONJG	Complex	Complex

*The function MOD or AMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2] a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as x .

Table 7-2. Basic External Functions

External Functions	Definition	Arguments	Name	Type of Argument	Type of Function
Exponential	e^a	1	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural Logarithm	$\log_e (a)$	1	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Logarithm	$\log_{10} (a)$	1	ALOG10 DLOG10	Real Double	Real Double
Trigonometric Sine	$\sin(a)$	1	SIN DSIN CSIN	Real Double Complex	Real Double Complex



SECTION 7
PROGRAMS AND SUBPROGRAMS

Table 7-2. Basic External Functions (continued)

External Functions	Definition	Arguments	Name	Type of Argument	Type of Function
Trigonometric Cosine	$\cos(a)$	1	COS	Real	Real
			DCOS	Double	Double
			CCOS	Complex	Complex
Hyperbolic Tangent	$\tanh(a)$	1	TANH	Real	Real
Square Root	$(a)^{1/2}$	1	SQRT	Real	Real
			DSQRT	Double	Double
			CSQRT	Complex	Complex
Arctangent	$\arctan(a)$	1	ATAN	Real	Real
		1	DATAN	Double	Double
	$\arctan(a_1/a_2)$	2	ATAN2	Real	Real
		2	DATAN2	Double	Double
Remaindering*	$a_1 \pmod{a_2}$	2	DMOD	Double	Double
Modulus		1	CABS	Complex	Real

*The function DMOD (a_1, a_2) is defined as $a_1 - [a_1/a_2] a_2$, where $[x]$ is the integer whose magnitude does not exceed the magnitude of x and whose sign is the same as the sign of x .



SECTION 8 OPERATING INSTRUCTIONS

8.1 GENERAL

The Varian Data Machines (ANSI) FORTRAN IV Compiler is available as a stand-alone program or as another processor within the Varian 620 Master Operating System (MOS). Whereas the **FORTTRAN IV** stand-alone program operates within a minimum configuration of 8,192 words of memory and a 33/35 ASR Teletype, the minimum for MOS FORTRAN IV is 12,288 words of memory plus the required MOS peripheral devices.

FORTTRAN programs and subprograms are compiled by the FORTRAN IV compiler. The FORTRAN IV loader loads main programs and all required subprograms into memory for execution. The FORTRAN IV run-time library provides the input/output, control, and mathematical functions required at execution time.

8.2 COMPILER OPERATING INSTRUCTIONS

The FORTRAN IV compiler translates **FORTTRAN IV** source programs to relocatable machine language programs in a single pass. Error diagnostics, source listings and object listings are provided. Input/output and listing options are selected for each program to be compiled. The memory reserved for integer and logical items can be selected as one or two words per item.

8.2.1 Stand-Alone FORTRAN IV

The Varian stand-alone FORTRAN IV compiler is supplied as an absolute binary object tape. The compiler is loaded into memory by the standard binary loader and occupies approximately 016650 words of memory. (See the reference handbook for the procedure for loading absolute object programs.)

The compiler is entered at location 0. Upon entry, the compiler executes a HALT with the P register set to 4, and the A register set to the upper limit of compiler-used memory (015777 standard for 8K). This limit can be modified by resetting the A register. To compile the programs, press RUN.

8.2.2 MOS FORTRAN IV

The initiation of the MOS FORTRAN IV compiler is accomplished by entering the control directive

/FORTRAN (or /F) P1,P2,...,Pn.



SECTION 8 OPERATING INSTRUCTIONS

This control directive directs the Executive program to call the System Loader to load the FORTRAN IV compiler and commence compilation. The parameter string specifies optional tasks that are to be performed. These options are:

- B No binary object program output desired.
- D Integer and logical items are assigned two words.
- L Load and go operation desired.
- M No memory map desired.
- N No source listing desired.
- O Octal listing of object program desired.
- X Conditional compilation desired. (Source records with an X in column 1 will be compiled.)

Input/output assignments during compilation are made through the /ASSIGN control directive (see 620 Master Operating System Reference Manual). The FORTRAN IV compiler uses the following logical units:

Source input	PI
Object output	BO
Listing	LO
Load and go	GO (optional)

8.3 LOADER OPERATING INSTRUCTIONS

8.3.1 Stand-Alone FORTRAN IV

The loader loads relocatable object programs produced by the FORTRAN IV compiler, and FORTRAN-compatible subprograms produced by the DAS 8A assembler. Object program input is from either paper or magnetic tape selected at the Teletype keyboard. Maps and error diagnostics appear on the Teletype printer.

The loader is on an absolute binary object tape and is loaded into memory by the binary loader. The FORTRAN loader occupies locations 000 through 007 and 0x4000 through 0x5740, where x is 0, 1, 2, 3, 4, 5, 6, or 7, depending on whether the computer memory size is 4K, 8K, 12K, 16K, 20K, 24K, 28K, or 32K words. (Locations 0200 through 0377 are



SECTION 8 OPERATING INSTRUCTIONS

reserved for loader-generated pointers.) The first program to be loaded must be a FORTRAN-compiled main program. Prepare the input unit, clear the registers and run at location STRT. A message then appears on the Teletype, requesting input selection. For paper tape input, type P. For magnetic tape input, type the unit number (0, 1, 2, or 3). If the selected unit is attached and ready, the loader loads the main program at 0500. The Teletype then types RQ, followed by a list of the required subroutines, followed by another input selection request. At this time, load all required subprograms.

For most efficient use of the loader, load standard FORTRAN library subprograms in the following order:

- a. Run-time input/output
- b. Complex math functions
- c. Double-precision math functions
- d. Single-precision math functions
- e. Double-precision math library
- f. Single-precision math library
- g. Run-time utility

Prepare and select the input unit as for main programs. The loader loads all required subprograms until an end-of-tape record or an end of file is detected, at which time the list of required subprograms is generated and input selection is again requested. Note that the end-of-tape record is not produced for subprograms by either the compiler or the assembler, but is supplied as a separate tape labeled FORTRAN END-OF-TAPE to be spliced to the end of the user's subprogram library tapes. Standard FORTRAN library tapes have end-of-tape records.

If two or more subprograms have the same name, only the first such subprogram input is loaded. When all required subprograms have been loaded, GO is typed followed by the load map, which lists each loaded subprogram with its entry point. To execute the main program, press RUN. Execution of the main program can be forced by running at location 0500.



SECTION 8 OPERATING INSTRUCTIONS

An error in loading outputs an error message and the loading map. A minus sign precedes the address of each subprogram that has not been loaded. In this case, the address is the last location where the subprogram is requested. All errors except checksum are nonrecoverable. Correct the error and reinitialize the loading process. Loading errors are listed below.

Error	Comment
CK	Checksum. Backspace the input tape one record and press RUN for another attempt.
AR	Area reference. An attempt has been made to load a value to an area not yet defined.
CE	Compiler error. A terminating error occurred at compilation time.
CS	Common size. A second use of blank common is larger than the initially defined area.
SZ	Program size. The program is too large for the available memory.

All FORTRAN main programs are loaded and entered at location 0500. Required subprograms are loaded as they are input to successive blocks of memory. Common storage overlays the FORTRAN loader. AID II routines and the absolute binary loader are in memory at their standard locations. Locations 0 through 077 are unused and locations 0200 through 0377 contain program and data pointers used by the program and subprograms to be executed.

To execute a FORTRAN program, initialize the input/output devices selected, clear the console registers, set the program counter to 0500, press SYSTEM RESET, and RUN.

There are three types of programmed halts: STOP, PAUSE, and EXIT. STOP causes a HALT 0777 with the stop number in the A register and -1 in the B register. STOP implies an end of job. PAUSE causes a HALT 0000 with the pause number in the A register, and zero in the B register. The program can be continued by pressing SYSTEM RESET and RUN. EXIT causes a HALT 0777 with -1 in the A and B registers, and signifies end of job. All programmed halts display error bits in the X register, as follows:



SECTION 8 OPERATING INSTRUCTIONS

Bit	Indication
0	Floating point overflow
1	Division check error —
2	Fixed point overflow
3	Indeterminate function
4	Log error —
5	Square root error
6	GO TO error

The following error halts are generated by the run-time I/O package. These errors give a four-character message on the Teletype, followed by a call to EXIT.

Message	Definition
FRMT	Format error
MODE	Data mode error (floating point vs. integer)
DATA	Input data field error
UNIT	Unit not attached or not available
TAPE	Checksum or tape parity error

All binary input/output records are a fixed length of 64 words. Paper tape records are punched with a record mark and checksum. Checksum errors encountered on input cause TAPE errors. Reading a file mark from magnetic tape causes a tape error halt.

BCD records can be fixed or variable length, depending on the device. However, only the first 80 characters are processed.

Keyboard and paper tape records are variable in length and are terminated by a carriage return followed by a line feed. The character ~ can be used to clear the input buffer and reset to column 1. Illegal characters are ignored. For keyboard input, the Teletype bell signals that BCD input is required. Maximum length of records is 72 characters for keyboard and 80 characters for paper tape. Card records are a fixed length of 80 characters. Illegal characters are treated as blanks.

Magnetic tape records are a fixed length of 84 characters, and should be card images with blank padding characters. Illegal characters are treated as blanks.

Note that the model 33 Teletype paper tape punch must be turned on and off by the operator.



SECTION 8 OPERATING INSTRUCTIONS

Line printer records are 120 characters. Printing is left-justified with unused positions set to blanks.

8.3.2 MOS FORTRAN IV

To run a program compiled under MOS FORTRAN IV, initialize the MOS, and load the compiled object program with the MOS directive (see Varian 620 Master Operating System Reference Manual, 98 A 9952 090):

`/LOAD (or /L)`

8.4 INPUT/OUTPUT DEVICE SPECIFICATIONS

8.4.1 Stand-Alone FORTRAN IV

For each program to be compiled a ? = will be typed on the teletype printer requesting input/output selection. The operator should respond by typing one of the following characters to indicate the input device.

C	Card reader
K	Teletype keyboard
P	Paper tape
0-3	Magnetic tape unit #

followed by one of the following characters to indicate the output device

C	Card punch
P	Paper tape
0-3	Magnetic tape unit #

followed by an optional listing selection character

S	Source listing
O	Object listing
B	Both source and object listings

followed by an item-sized character

blank	One-word integer and logical items
D	Two-word integer and logical items

followed by the character >, followed by the one- to six-character program name (optional), followed by @ for a carriage return and line feed. For example

? = CPSD > MATRIX @



SECTION 8 OPERATING INSTRUCTIONS

where

- C indicates card input
- P indicates paper tape output
- S indicates source listing of the program named MATRIX
- D indicates two-word integer and logical items

Following input/output selection, source statements are read and object records are output through the selected devices. Error diagnostics and selected list options are printed on the teletype or line printer (if available). Upon detection of an END statement followed by a nonblank statement, the compiler produces a program map listing all variables, constants (in octal), and required subprograms. After listing the program map the compiler types a ? = to permit compilation of another program.

8.4.2 MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual (98 A 9952 090).

8.5 COMPILER INPUT RECORDS

8.5.1 Stand-Alone FORTRAN IV

Input to the compiler is a series of **FORTRAN** statements, each of which appears in one or more input records. Records can be fixed or variable length depending on the device; however, only the first 72 characters of each record are used by the compiler. Any illegal characters are treated as blanks. END statements must be followed by at least one nonblank record (another END statement is suggested).

Keyboard and paper tape records are variable length and are terminated by a carriage return and line feed in that order. The character > can be used to TAB to column 7, and the character - can be used to clear the input buffer and reset to column 1. For keyboard input the teletype bell is used to notify the operator that source input is required. Position the paper tape so that the first character is at the read station.

Card records are a fixed length of 80 characters. Magnetic tape records should be card images.

8.5.2 MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual (98 A 9952 090).



SECTION 8 OPERATING INSTRUCTIONS

8.6 COMPILER OUTPUT RECORDS

8.6.1 Stand-Alone FORTRAN IV

Object records are output as they are created. Paper tape object programs are punched with leader and trailer records.

Magnetic tape object programs are terminated by an end of file and a backspace to permit the stringing of compiled subprograms. All main programs are terminated by an end-of-tape record.

All error diagnostics are of the form

ERR xx a...a

where

xx is a number from 1 to 18 (notification error), or T1 to T9 (termination error)

a...a represents the last (up to 12) characters encountered in the statement being processed.

The rightmost character indicates the point where the error was discovered (the character - indicates end of statement). If a termination error is discovered, object output is terminated, but source code is continued to detect any further errors.

8.6.2 MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual (98 A 9952 090).

8.7 NOTIFICATION ERRORS

8.7.1 Stand-Alone FORTRAN IV

- 1 Construction
- 2 Usage
- 3 Mode
- 4 Illegal DO termination
- 5 Improper statement number
- 6 Common base lowered
- 7 Illegal equivalence group
- 8 Reference to nonexecutable statement



SECTION 8 OPERATING INSTRUCTIONS

- 9 No path to this statement
- 10 Multiply defined statement number
- 11 Invalid format construction
- 12 Spelling error
- 13 Format with no statement number
- 14 Function not used as variable
- 15 Truncated value
- 16 Statement out of order
- 17 More than 29 *COMMON* regions
- 18 Non*COMMON* data

8.7.2 MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual (98 A 9952 090).

8.8 TERMINATION ERRORS

8.8.1 Stand-Alone FORTRAN IV

- T1 Construction
- T2 Usage
- T3 Data pool full
- T4 Illegal statement
- T5 Improper use of name
- T6 Improper statement number
- T7 Mode
- T8 Constant too large
- T9 Improper *DO* nesting

8.8.2 MOS FORTRAN IV

See the Varian 620 Master Operating System Reference Manual (98 A 9952 090).

8.9 OPTIONAL LISTINGS

Source and object records can be listed. Source records are listed as they are input. Object records are listed as they are created. Each object record consists of a varying number of two- and four-word data/instruction entries, one listing line per each entry. Two-word entries are of the form



SECTION 8 OPERATING INSTRUCTIONS

abbc wwww

and four-word entries are of the form

abbc nnnnnn wwww

where

a	is the control code
bb	is the subcode
c	is the pointer number
nnnnnn	is a one- to six-letter subprogram name
wwwvv	is a six-digit octal value or instruction

8.10 MAPS

8.10.1 Stand-Alone FORTRAN IV

8.10.1.1 RUNTIME

See figure 8-1.

8.10.1.2 PROGRAM

Upon processing the END statement, the compiler lists the program map. The first three lines of the map define the size of the program, data and common areas. They are of the form

a *SIZE mmmmmm

where

a is the area (0 = program, 1 = data)
mmmmmm is the actual size

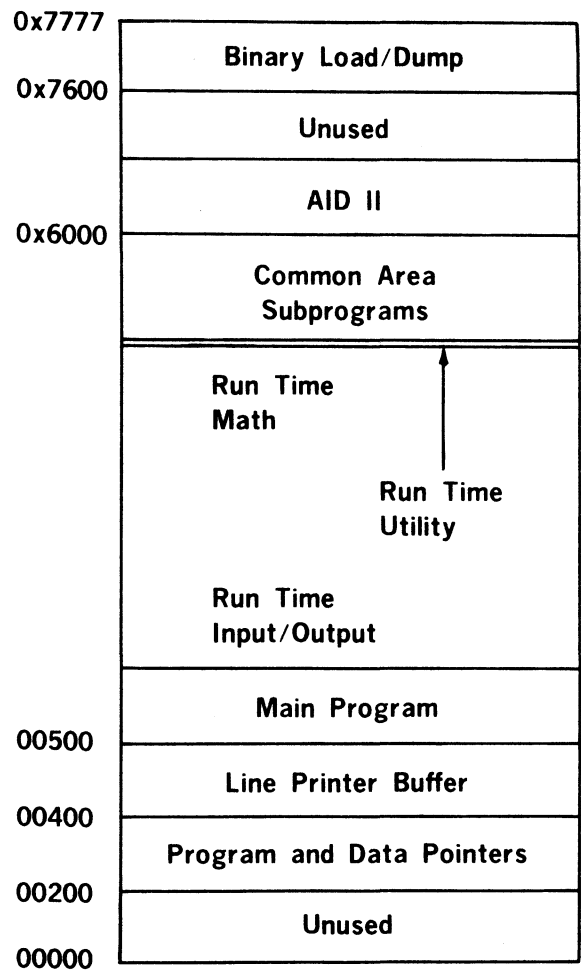
For programs with no termination errors, the following information is also listed.

a,	rrrrr	nnnnn	Variable
a,	rrrrr	cccccc ccccc	Constant
S,	rrrrr	nnnnn	External subprogram
#,	rrrrr	ssss	Statement number
X,	rrrrr	ssss	Undefined statement number
c,	rrrrr	nnnnn	COMMON variable



SECTION 8
OPERATING INSTRUCTIONS

B.3 EXECUTION TIME MEMORY MAP



VTIF-0996A

Figure 8-1. Run-Time Map for Stand-Alone FORTRAN IV



SECTION 8 OPERATING INSTRUCTIONS

where

a	is the area
rrrrr	is the relative location of the item or the last reference to the subprogram or statement number
cccccc ccccc	is a two-word octal constant
ssss	is a statement number

8.10.2 MOS FORTRAN IV

8.10.2.1 RUNTIME

See figures II-1 and II-2 in the Varian 620 Master Operating System Reference Manual (98 A 9952 090).

8.10.2.2 PROGRAM

In the following sample printout of an MOS FORTRAN IV program map, the leftmost six-digit column is the value of the relative location counter of the item to the right. The letter and name, value or variable identifies the item as a relocatable (R), fixed, or floating-point value or variable, or a subroutine entry name, the name of an External (E) subprogram or the name of a region in COMMON (C). COMMON is the name for blank or unlabeled COMMON under MOS.

Sample Program Map

ENTRY/COMMON BLOCK NAMES

```
000043 R TEST1
000004 C COMMON
000004 C      XX
```

EXTERNAL NAMES

```
000020 E      $SI
```

SYMBOL TABLE

```
000023 R 000001
000027 R 000002
000000 C      A
000002 C      B
000000 C      C
000002 C      D
000025 R      I
000005 R      5
```



SECTION 8 OPERATING INSTRUCTIONS

```
000031 R      J
000035 R      K
000033 R 000003
000012 R      10
000041 R      R
000037 R 040306 031463
000020 E      $ST
```

8.11 I/O DEVICE SELECTION

8.11.1 Stand-Alone FORTRAN IV

The following logical unit numbers are associated with the indicated devices at execution time.

Unit Number	Assignment
Logical unit 0	Teletype keyboard and printer
Logical unit 1	Teletype paper tape reader and punch
Logical unit 2	High-speed paper tape reader/punch
Logical unit 3	Card reader/punch
Logical unit 4	Line printer
Logical unit 8	Magnetic tape unit 0
Logical unit 9	Magnetic tape unit 1
Logical unit 10	Magnetic tape unit 2
Logical unit 11	Magnetic tape unit 3



SECTION 8 OPERATING INSTRUCTIONS

8.11.2 MOS FORTRAN IV

The logical unit names corresponding to the unit numbers, u, in READ, WRITE, and auxiliary I/O statements are:

Number	Name	Definition
1	SF	System file
2	SI	System input
3	SO	System output
4	PI	Processor input
5	LO	List output
6	BI	Binary input
7	BO	Binary output
8	SS	System scratch
9	GO	Load and go
10	PO	Processor output
11	S1	Scratch #1
12	S2	Scratch #2
13	S3	Scratch #3
14	S4	Scratch #4
15-255	None	User-definable files



APPENDIX A

Symbol	620 Internal ASCII	Standard BCD Codes			
		Line Printer	Mag Tape	Punched Card	FORTRAN Internal
@	300	00	32	0-2-8	72*
A	301	01	61	12-1	13
B	302	02	62	12-2	14
C	303	03	63	12-3	15
D	304	04	64	12-4	16
E	305	05	65	12-5	17
F	306	06	66	12-6	20
G	307	07	67	12-7	21
H	310	10	70	12-8	22
I	311	11	71	12-9	23
J	312	12	41	11-1	24
K	313	13	42	11-2	25
L	314	14	43	11-3	26
M	315	15	44	11-4	27
N	316	16	45	11-5	30
O	317	17	46	11-6	31
P	320	20	47	11-7	32
Q	321	21	50	11-8	33
R	322	22	51	11-9	34
S	323	23	22	0-2	35
T	324	24	23	0-3	36
U	325	25	24	0-4	37
V	326	26	25	0-5	40
W	327	27	26	0-6	41
X	330	30	27	0-7	42
Y	331	31	30	0-8	43
Z	332	32	31	0-9	44
[	333	33	75	12-5-8	73*
\	334	34	36	0-6-8	74*
]	335	35	55	11-5-8	75*
!	336	36	17	7-8	76*
-	337	37	20	2-8	77**
blank	240	40	20	No punch	00
!	241	41	52	11-2-8	51
"	242	42	35	0-5-8	62
#	243	43	37	0-7-8	63
\$	244	44	53	11-3-8	60
%	245	45	57	11-7-8	64



APPENDIX A

Standard BCD Codes (continued)

Symbol	620 Internal ASCII	Line Printer	Mag Tape	Punched Card	FORTRAN Internal
&	246	46	77	12-7-8	65
'	247	47	14	4-8	66
(	250	50	34	0-4-8	52
)	251	51	74	12-4-8	53
*	252	52	54	11-4-8	47
+	253	53	60	12	45
,	254	54	33	0-3-8	54
-	255	55	40	11	46
.	256	56	73	12-3-8	51
/	257	57	21	0-1	50
0	260	60	12	0	01
1	261	61	01	1	02
2	262	62	02	2	03
3	263	63	03	3	04
4	264	64	04	4	05
5	265	65	05	5	06
6	266	66	06	6	07
7	267	67	07	7	10
8	270	70	10	8	11
9	271	71	11	9	12
:	272	72	15	5-8	67
;	273	73	56	11-6-8	70
<	274	74	76	12-6-8	56*
=	275	75	13	3-8	55
>	276	76	16	6-8	57***
	277	77	72	12-2-8	71*

* Undefined character

** Form control: return to column 1 (undefined in MOS)

*** Tab control: skip to column 7 (undefined in MOS)



APPENDIX A

Symbol	620 Internal ASCII	Standard BCD Codes			
		Line Printer	Mag Tape	Punched Card	FORTRAN Internal
@	300	00	32	0-2-8	72*
A	301	01	61	12-1	13
B	302	02	62	12-2	14
C	303	03	63	12-3	15
D	304	04	64	12-4	16
E	305	05	65	12-5	17
F	306	06	66	12-6	20
G	307	07	67	12-7	21
H	310	10	70	12-8	22
I	311	11	71	12-9	23
J	312	12	41	11-1	24
K	313	13	42	11-2	25
L	314	14	43	11-3	26
M	315	15	44	11-4	27
N	316	16	45	11-5	30
O	317	17	46	11-6	31
P	320	20	47	11-7	32
Q	321	21	50	11-8	33
R	322	22	51	11-9	34
S	323	23	22	0-2	35
T	324	24	23	0-3	36
U	325	25	24	0-4	37
V	326	26	25	0-5	40
W	327	27	26	0-6	41
X	330	30	27	0-7	42
Y	331	31	30	0-8	43
Z	332	32	31	0-9	44
[	333	33	75	12-5-8	73*
\	334	34	36	0-6-8	74*
]	335	35	55	11-5-8	75*
!	336	36	17	7-8	76*
-	337	37	20	2-8	77**
blank	240	40	20	No punch	00
!	241	41	52	11-2-8	51
"	242	42	35	0-5-8	62
#	243	43	37	0-7-8	63
\$	244	44	53	11-3-8	60
%	245	45	57	11-7-8	64

Fold

FIRST CLASS
PERMIT NO. 323
NEWPORT BEACH,
CALIFORNIA

BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES



varian data machines / a varian subsidiary
2722 michelson drive / irvine / california / 92664

ATTN: TECHNICAL PUBLICATIONS

Fold



varian data machines / a varian subsidiary